UiO : Department of Informatics
University of Oslo

Developing a web-based tool for analysing cell type-specificity of genomic variation data

Kristoffer Waløen

Master's Thesis, Spring 2013

Developing a web-based tool for analysing cell type-specificity of genomic variation data

Kristoffer Waløen

May 2, 2013

Abstract

The majority of trait associated variants found in GWAS studies lie within non coding sequences. This suggests that a large proportion of variants alter regulatory regions. Certain genomic features has been shown useful as marks of cell type specific activity of genomic regions. Analyzing such genomic features against variant regions may therefore be used to find previously unknown links between trait and cell type. Although there have been done several investigations of this type, no easily accessible tools for this type of research exists. This makes reproduction of such results difficult and time consuming, hindering confirmation and updates of such results

Such an accessible tool for studying cell-type specificity of genomic regions is presented here, created in a Galaxy-based web interface at the Genomic HyperBrowser server. It allows the user to run a selection of analyses on their own genomic variation data against genomic tracks of cell-type specific marks. A table presenting the main results provides a broad overview of the most relevant cell types, while links to further details behind each main result allows for deeper investigations.

The tool here presented allows anyone to run such analyses without deep knowledge of statistics and informatics, as most parameters and variables are set automatically by the system. Combined with the graphical interface in the HyperBrowser, this makes it easy to specify and reproduce analyses.

Contents

List of Figures

Listings

Acknowledgments

First of all I would like to thank my supervisor *Geir Kjetil Sandve* for the help and guidance through this thesis, and for leading the weekly meetings. The second one who deserves to be thanked is *Nils Damm Christophersen*, for his friendly tone and willingness to share of his vast experience.

Then I want to mention those who have taken the time and effort to give me feedback on structure and language. *Oda Josefine Noven* for taking the time to read an early draft and give feedback. *Erica Brooks* and *Allan Ingalls* deserves a big thanks for going through this thesis correcting the errors in my spelling.

I also want to thank all the master students at the 10th floor at *Ole-Johan Dahls hus* at UiO for the good atmosphere, lunch breaks, knowledge sharing, laughs and of course the optimizing of coffee brewing.

The final shout out goes to everyone else who have had to accept me saying no to all the fun things lately.

Kristoffer Waløen
University of Oslo
May 2013

Part I

Introduction and background

Chapter 1

Introduction

Deoxyribonucleic acid (DNA) can be seen as the blueprint of life. It is DNA, together with environmental factors, that designs you as human, and just as most humans are similar, so is the DNA between humans much the same. But there are differences. Some of these differences are more or less harmless, eye color, hair color and skin color are all traits that to some extent are determined by the DNA [27]. Cancer, Multiple Sclerosis and Alzheimer are traits with more serious consequences that also have been linked to variations in the DNA [14][13][40]. Genome-wide association study (GWAS) are among the techniques used to connect genetic variations to traits.

There have been much work done within the field of GWAS, and more and more genetic markers for diseases and traits are being mapped and collected. In 2010 more than 50% of the papers published in Nature Genetics was GWAS related [22]. Publicly available databases contain huge amounts of information, about markers for cell types, and markers for traits and diseases. Much work is carried out with this data, but a lot of the work is done on custom-made computer software or on a selection of different tools[5]. This can make it time consuming and difficult to reproduce results.

Creating a custom made computer software for analysing genetic data, including GWAS data, can be very time consuming, especially if including the time it takes to assure correctness and improving readability. Time that otherwise could have been spent doing research, is used to create the tools needed for the research. By creating a publicly available tool with a graphical interface that is easy to use, it might be possible for the scientists to save time by being allowed to focus on research and not software development.

1.1 Genomic variation analysis tool

The basic idea is to create a tool that allows the scientists to do statistical analyses on genomic data against given genetic regions, without having to worry about all the statistical variables. It is to do statistical analyses and comparison on genetic data for a trait against user chosen genetic regions.

The results should be presented so that it is easy to see which of the analyses that gave interesting results, and where it could be interesting to take a closer look at the data.

The goal is that it must be easy for the user to choose what to run the analyses on. They should have access to pre-stored trait data, but also be allowed to upload their own research data. They should be allowed to choose between a few but relevant genetic regions to run their data against, and they should be allowed to choose more than one region. These genetic regions will contain subtypes, these subtypes are the phenotyped regions. One example of this is the chromatin state active promoter as genetic region, and b-cells as one of the subtypes.

The user should have a few but relevant analyses to choose from, also here it should be possible to select more than one. Relevant analyses will tell if there are any relations between the genetic data.

The tool will be implemented in the Galaxy-based Genomic HyperBrowser, which gives a web-based graphical user interface and tools for publishing and reproducing results.

1.1.1 Flexibility

The goal is to make the tool flexible, but still easy to use. It should let the user do some selections, but is should not bog down the user with too many choices. The tool should only show the necessary selections at all times. This way the user only has to focus on what is needed for that specific run, while still being allowed some individual selection. The main focus will be on ease of use.

1.1.2 Presenting the results

When all analyses are done, the results should be presented in a way that makes it easy for the user to get an overview of the results. Since the user should be allowed to run several analyses on several genetic regions, it can not present all information up front. Some information must be selected, and presented as the main result. This information will have to be representative for the local results for that subtype. Graphs should also be presented for at least some of the analyses.

1.1.3 Checking the result

The final step for this tool will be to do some analyses on genetic variations for MS, and then confirm the relations between MS and B cells, as found in “Genomic Regions Associated with Multiple Sclerosis Are Active in B Cells” [13]

1.2 Genomic track overlap algorithm

In the HyperBrowser there are various algorithmic implementations to compute the number of base pairs overlapping between two genomic

tracks, depending on properties of the tracks and how the data is represented. The analyses involved in this project relied heavily on the computation of overlap between genomic tracks, where there could be intra-track overlap between segments of one of the two tracks.

As existing algorithms for this situation were either slow or had particular requirements on how the data were represented, it was decided to develop an improved algorithm for computing base pair overlap between two tracks, where one can have intra overlapping segments.

1.3 Accessibility

The tool, *Analyze TF- or cell type-specificity of a genomic track*, is publicly available at http://hyperbrowser.uio.no/personal/, under the *Restricted and experimental tools* link. It is publicly available for anyone interested, with some histories available under *Shared Data* and *Published Pages*.

A copy of all the files can be found at http://hyperbrowser.uio.no/dev2/static/downloads/kriswalo-files.tar.gz.

Chapter 2

Background

This chapter will go through useful background information. It will cover fields such as genetics, which is the field bioinformatics is working on. Then bioinformatics will be explained, bioinformatics is one of the fields that work with data collected from Genome-wide association study (GWAS). Then it will be taken a closer look at personal medicine, which might be said to be one of the fields benefiting from the work done in GWAS and bioinformatics. Then the focus will shift to the more technical aspects. It will be taken a closer look at the programming language python, which is the language the Genomic HyperBrowser is written in. The Genomic HyperBrowser is the framework the tool is built within, and will be described in the section after Python. The final technical aspect that is covered in this chapter is legacy code, as the HyperBrowser is a good example of legacy code. The last section of this chapter will go through a set of articles containing important information that has been used in this work.

2.1 Genetics

DNA and RNA are known as nucleic acids, and DNA contains four nucleotides. Nucleotides are the building blocks of both DNA and RNA [6], and when the nucleotides are linked together in a linear manner they form a strand [6]. Two strands can be bound together to create a double helix [6]. When DNA is associated with proteins into a compact structure it is called chromatin, while if it is an array of different proteins it forms chromosomes [6]. One nucleotide contains only one of five possible bases: adenine (A), guanine (G), cytosine (C), thymine (T) and uracil (U), where thymine is unique for DNA, and uracil for RNA [6]. The AT/CG rule says that if you know the sequence of one strand, you also know the sequence of the opposite strand. For example, if one strand is GCGGATT, then the opposite strand has to be CGCCTAA [6].

DNA replication is a process in living cells, a process that needs speed, accuracy and that must avoid leaving gaps in the newly made strands [6]. It is very accurate, with less than one mistake for every 10^9 base pair copied [30]. But when such a permanent error occurs, it called a

mutation [6].

A gene is a linear sequence of amino acids that encodes a polypeptide, where polypeptide means structure [6]. A protein, on the other hand, means function, and a protein can be created from one or more genes [6]. The process from gene to polypeptide contain several steps, but can be roughly summarized in two, transcription and translation. The transcription creates a RNA copy of a gene, more specifically messenger RNA (mRNA) [6]. The mRNA is then transported to the ribosome, which then translates the mRNA into a amino acid sequence of a polypeptide [6]. Two other types of RNA are transfer RNA (tRNA), which translates the mRNA, and ribosomal RNA (rRNA), which forms a part of ribosomes [6]. This means that for some genes the RNA is the product in itself [6]. Ribosomes are composed of many proteins and rRNA, and provides a location where mRNA and tRNA can interact [6]. Generally the transcription starts near a site in the DNA called the promoter, and the terminator signals the end of the transcription, while the regulatory sequence either increase or inhibit the rate of transcription [6].

Normally DNA is tangled together around histones (proteins), but usually there are strains of DNA less tangled together. These strains are usually genes more sustainable to transcription and are called DNase hypersensitive sites (DHS). DNase is enzymes that digest DNA[6]. These sites may have different positions depending on what cell type the gene is working within.

A mutation in a coding sequence may not affect the function of the polypeptide or protein, it is then called a neutral mutation [6]. While a nonsense mutation is a mutation that makes the translation end to early, this often leads to polypeptides not functioning as normal [6]. A frameshift mutation adds or deletes nucleotides that are not in multiple of three, this shifts the reading frame so that a completely different amino sequence occurs. This is likely to inhibit the protein function [6]. A mutation may also occur in noncoding sequences, and in that way increase or decrease the rate of transcription [6]. Proteins that control the ability to transcribe genes is called transcription factors (TF), hence the position will be cell specific [6]. H3K4me3 is a protein working as promoter for active transcription sites [31].

Phenotype is how the genes get expressed [6].

2.2 Bioinformatics

Bioinformatics is the systematic analysis of information relating to biological macromolecules with the aid of computers [42]. Macromolecules are large molecules such as nucleic acids, lipids, proteins and carbohydrates. The early fundamentals of bioinformatics can be traced all the way back to the 1960s, but the term bioinformatics did not exist at that time [42]. In the 1980s the GeneBank was established, which is a database for collecting all publicly available DNA sequences. And with it came algorithms for fast searching in databases, such as FASTA and BLAST [42]. But the main

reason bioinformatics gained popularity was the advancement of genome studies that now produces huge amounts of biological data [42].

Bioinformatics is a term used for the research field where biology meets computer science [42]. Many definitions exists, the term can be used for research that uses computers for storage, retrieval, manipulation, and distribution of information related to biological macromolecules [42]. The broader term computational biology also includes biological areas that includes computations, such as mathematical modelling of ecosystems, population dynamics and application of game theory in behavioral studies [42]. But there does exist other definitions as well, and hence these should not be seen upon as the only correct definitions.

One could say that the ultimate goal in bioinformatics is to better understand the living and how it works at the molecular level [42]. Bioinformatics can help us understand this by analysing and simulating how DNA becomes proteins [42].

The development of computational tools, and the applications of these tools to get a better understanding, can be said to be two subfields of bioinformatics[42]. These two fields are very much dependent on each other, you need the tools created to study the genomic data we have. As we learn more, new questions arise, and new tools are needed.

It is important to recognize the limitations of bioinformatics. The results are not formal proof of concepts, and they do not replace traditional research [42]. There are many places an error may arrive, and start to snowball. Bioinformatics is no better than the data it is based upon, and sequence data often contain errors [42]. It is also important to remember that bioinformatics often works with huge amounts of data, which means that the algorithms have to make a choice between accuracy and speed, while at the same time algorithms lack the capability to completely and truthfully reflect real life [42]. Hence it is important to run research on several types of data if available and with several different types of algorithms to find a consensus by comparing all the results [42].

2.3 GWAS

2.3.1 GWAS introduction

GWAS is short for genome-wide association study and is the search for genetic variants across the human genome in an effort to identify genetic risk factors for diseases that are common in the population [7]. Genetic variants can be in the form of SNPs, or less common mutations. These genetic variants can then be used for many applications in research and clinical practice of medicine, such as preventive medicine, personalized medicine, pharmacogenomics and pharmacogenetics. One of the most successful fields within GWAS has been pharmacology [7]. In pharmacogenetics the goal is to identify DNA sequence variations that are associated with drug metabolism and efficiency as well as adverse effects [7]. One example is warfarin (see section 2.4.2).

A genetic variation in a single allele that occurs with a high frequency in the population is called a single nucleotide polymorphism or SNP [7]. SNPs are typically used as genomic markers for a genomic region, with the large majority of them having minimal impact on biological systems. SNPs can have different functional consequences such as causing amino acid changes, changes to mRNA transcript stability and changes to transcription factor binding structure (that suggests common origin) [7]. Within a population a SNP typically has two commonly occurring base-pair possibilities for a certain positions. The frequency of a SNP is given by the frequency of the less frequent allele [7]. This means that if a SNP with the minor allele (G) has a frequency of 0.3, that implies that 30% of the population has this G allele, compared to the more common allele, which then is found in 70% of the population. Very rare diseases can be caused by genetic variants with very low frequency, and the genetic variants is then often referred to as a mutations, although they can be structurally equivalent to SNPs [7]. In genetic literature the term SNP is generally used to common single base-pair changes, and mutation to rare genetic variants. If a SNP has 30% frequency in the population, and almost directly leads to a disease phenotype, almost 30% of the population would have that disease [7]. Then the allele frequency and the spread in the population would be almost correlated. But if the SNP causes a small change in gene expression that just changes the risk for a disease by a small amount, the prevalence of the disease and the allele frequency would only be slightly correlated. Hence common variants almost by definition cannot have high penetrance [7]. It has been found that around 40% of trait-associated SNPs fall in intergenic regions, and another 40% are in noncoding introns [28]. These results have made the role of intronic, and particularly intergenic, regions to gene expression interesting. Another interesting result from GWAS is the finding of SNPs related to a disease in genes earlier not believed to have any connection to that certain disease [28].

When embarking upon a genetic study, the initial focus should be on identifying precisely what quantity or trait the genetic variations influences. The most common design for GWAS is the case control design. In the case control design the allele frequency in patients with a certain trait is compared to patients without the trait. This design is often easier and less expensive than studies using other designs [33]. This design also carries the most assumptions, and if mistaken, can lead to severe biases and wrongly made associations [33]. One of the most important biases is that of the unrepresented case participants. These participants are typically gathered from clinical sources and therefore may not include fatal, mild or silent cases not coming to clinical attention [33]. It may also be difficult to ensure the comparability of the case and control participants, who may differ in important ways that could effect the genetic risk factors [33].

2.3.2 History

In 2000 a collaborating project was started up, the project's mission was to identify up to 150 000 SNPs throughout the human genome within two

years, and to make the information available to the public [22]. In April 2003 the sequencing of the human genome was announced completed. 99% of the gene-containing part of the human sequence was finished with 99.9% accuracy. But the human genome sequence has still not been finished, there are still many gaps in the reference sequence. And given the variation of the human genome, one might always ask what is the completion of the human genome [22]. The idea behind the HapMap project was to create a public genome-wide database of common human sequence variations, providing information needed as a guide to studies of clinical phenotypes. The researchers created a database of 1 million genetic variations for 4 representative populations, Africans, European Caucasian, Chinese and Japanese [22].

This made GWAS easy to do, just collect DNA from patients, genotype it, and then do some statistical analyses on the results. Genotyping became work, not research, and commercial genotyping systems have spread all over the world. By sharing control groups, collecting them is no longer necessary, except for diseases with extraordinary high prevalence(>10%) [22]. As a consequence of HapMap, commercial genotyping systems, such as Illumina, have spread out around the world, and with them, GWAS as well. In 2007 the study of genetic polymorphisms (SNPs) and its applications were selected for the No. 1 scientific event of the year [22]. And as mentioned previously, 56% of 181 published papers in Nature Genetics was GWAS related [22]. Some of the diseases that have been linked to genes are obesity, hypertension, diabetes and osteoporosis [22].

2.3.3 Common problems

A frequent complaint against GWAS is that the results mean little to patients due to the small effect of variants on disease risk [22]. For example, one paper reported that a certain SNP was associated with height, but its estimated additive effect is only 0.44 cm on height [37]. Another example is a GWAS study that found that a certain allele is related to the cartilage thickness of the hip joint measured on X-ray films. Those with the allele had a 0.07 mm narrower joint [8]. This shows that GWAS is not a conclusive method, it is a method to map the gene; an associated variant is not necessarily a true casual variant [22]. Another problem comes from the controls with ambiguous definitions of populations, such as "UK Caucasians" or "Western Europeans". This results in a compromised result by adding a mixture of different ethnical subgroups. Hence it is important to detect population layering prior to analysis [22]. It is important that statistics leads to biology, and biology to medicine. There are still many more genes to be identified. The gene DVWA, which has been associated with osteoarthritis, and was found in what was thought to be a gene desert region when found in 2004 [22]. There is also a big chance of error in the grouping of individuals "case" or "control". Take for example the disease multiple sclerosis which is a complex disease often diagnosed over a long period of time by ruling out other possible conditions, here individuals could easily end up in the wrong group. Despite this loose classification

of case and control, GWAS on multiple sclerosis have been very successful, implicating more than 10 new genes for the disorder [7]. To predict the chance of getting a complex disease, genotypes at multiple SNPs are often combined into scores calculated according to the numbers of risk alleles carried. From what is seen so far it is obvious that genetically based risk assesment of disease occurrence is too uncertain to be of any clinical predictive value. But that might change as the sample size increase and more risk variants are identified [28]. A huge challenge for the physicians and patients in years past and years to come is correct communication of risk prediction, because the perception of risk is often more heavily influenced by emotion than science [28].

Another problem within GWAS is the false positives [22]. With $P < 10^{-3}$ we have 1 in 1000 coincidence. Or if you try 1000 times, you can hit 1 target and 100 000 times, 100 targets by chance. Consequently it is important with replication of studies, but not on the same population, as it may duplicate the same errors as in the previous study [22]. For this reason replication in large samples are the best way to duplicate associations [22].

GWAS can easily be seen upon as a way of identifying markers for genetic identifiers and risk assesment. By knowing the genetic risk we can take steps towards minimizing other risk factors through our way of life. Also by studying the meaning of allelic difference of causal variants, it is possible to clarify the molecular development of the disease. And with that knowledge create innovative treatments and discover new drugs [22].

2.4 Personalized medicine

2.4.1 Introduction

As mentioned in section 2.3 both personalized medicine and avoiding adverse drug effects can be among the goals of GWAS. And both of these can be put under the category personal medicine. Personal medicine is a wide ranging subject with many topics. To define it in one sentence is difficult. And there is no general consensus on what personal medicine is, only that it is the future [26]. It is not a question about if, but when. Personal medicine may be considered as finding the most efficient treatment based on genetic variations. Or it may be used to increase drug safety, by maximizing the health benefit and minimizing the chance of adverse drug effects. But this is not the only topics under personal medicine. Personal medicine can also be seen as preventing, diagnosing, treating and optimizing individual health-care decisions [26].

2.4.2 Current status

One field within personal medicine focuses on finding molecular biomarkers for drugs. This is usually done post hoc, by looking at the data after the trials for a new drug have been concluded. Three examples that used post hoc analysis are: abacavir, used against HIV and Aids; irinotecan, used against cancer; and warfarin, used against thrombosis (information can be

found at www.drugbank.ca). It can be argued that it is better to co-develop drugs and tests for biomarkers, as have been done with trastuzumab, used against breast cancer. They created a diagnostic tool concurrently with the drug, and got it approved and recommended by the FDA. FDA is the Food and Drug Administration in the US, they have to approve the drugs before they can hit the US market. Similar administrations exists in most countries and the EU. Creating the diagnostic tools concurrent with the drug might help targeting user groups that the drug has the desired effect on. The tool can then be presented to the FDA as a selection criterion, and hence increasing the chances of approval. The diagnostic tool can also be used to remove selected groups with increased possibility of adverse drug effects. Hence also increasing the chances of approval, while decreasing the chance of ending up like Rofecoxib. Rofecoxib was originally approved in 1999 as a drug against inflammations, but it was discovered to drastically increase the chance of heart attack, and thus removed from the market in 2004. Sponsors of drug development seem to think that it is common to co-develop diagnostic tools and drugs, but in reality co-development rarely happens due to uncertainty about the economical benefit [10].

Currently there is a productivity problem in the pharmaceutical industry [26]. Although the research and development has increased drastically, there has not been an increase in approved drugs. There are many theories as to why this is the case, but one thing is clear, the old business model is at least partly responsible. The pharmaceutical companies have been looking for groundbreaking drugs, drugs that fit everyone and work well. These drugs have high development cost, but have so far paid for them self by hitting a big need in the market and hence selling huge amounts. Rofecoxib was one such drug, selling for more than $2.5 billion per year until was withdrawn due to side effects [26]. As more and more of these drugs have been discovered, it's gotten harder and harder to find new ones. And the question now asked is, can this business model be maintained? There seem to be a change within the pharmaceutical industry towards drugs that works well on some, and to find that group they want to use genomic biomarkers. This will increase the capital to a section of personal medicine, and might very well be something that very well speed the arrival of personal medicine.

2.4.3 Genetics directly to consumer

As mentioned in section 2.3.3, one of the problems physicians face is to correctly communicate risk assessments. The companies within genetics directly to consumers, are trying to sell this type of information to consumers.

So far more than 100 000 individuals have tested their DNA against the more than 1000 DNA variants associated with diseases [23][32]. The direct-to-consumer, DTC, industry sends the consumer a small package for sampling the DNA, and within a few weeks, they present the absolute risk for certain diseases. Absolute risk is calculated from relative risk and average population risk, where relative risk is based on the individual

genetics, while average population risk is based on the population. The average population risk variable will change depending on how you define population and hence there may be some deviation between the different companies. But even in relative risk there are differences to be found. In [32] they sampled the DNA of five persons, 3 females and 2 males, and sent them to Navigenics and 23andMe to test against 13 diseases. They found that the two companies on some of the diseases had different population risk and relative risk. This despite that the companies agreed on more than 99.7% of the DNA bases observed [32]. So why do they not agree on risk predictions? For one, Navigenics separates between men and women on population risk, while 23andMe uses age. Another reason is that they do not use the same set of markers when it comes to calculate relative risk, and because of that the results will vary [32]. This implies that the companies will not always agree on the absolute risk of a certain disease. It was shown that for 7 diseases they found that 50% or less of the predictions agreed across the five individuals [32]. On Crohn's disease the two companies varied from high risk to low risk on two of the five candidates, and between average and low on the third. This highlights what was mentioned in section 2.3.3 that GWAS has had very little impact on patients and that there is still some uncertainty about the direct use of it, although some companies are already selling products using this information.

2.5 Python, the language of choice

It is practical to divide different programming languages into two categories: typesafe languages and dynamically typed languages. Typesafe languages bind variables to certain data types, while dynamically allows variables to hold any type of data [25]. Another typical way of classifying the languages is by defining them as high level or low level languages. High and low does not represent quality, but how they are written. High level languages are close to natural language specifications of algorithms, while low level languages are closer to hardware level. Python and Perl are examples of high level languages, while C and Fortran are low level. C++ and Java would then end up somewhere in between [25]. It is worth noting that there are no strict borders between high and low level languages. There is also a question about how low the high level languages go, and how high the low level languages go.

There are mainly two reasons for Python's success in scientific computing. First of all, Python typically gives readable and concise code, which in return makes for faster development cycles. Secondly there is Python's access to its internals from C via the Python/C API [4]. Python's role in scientific computing is usually binding together existing components instead of reinventing the wheel. For example, SciPy, a Python specific package, contain more than 200 000 lines of C++, 60 000 lines of C, and 75 000 lines of Fortran, compared to about 70 000 lines of Python code [4].

```
1 F = open(filename, 'r')
2 n = F.read().split()
```

Listing 2.1: Reading elements from a file into a list in Python.

2.5.1 Efficiency

The high level and flexibility of Python will necessarily imply some loss of efficiency, especially when traversing big data structures. However, with modern computers, it may often be fast enough. A factor of 10 times slower code might not really matter when the statements in the script is executed in a few seconds or less [25]. In some cases the high level language will give near optimal efficiency. An example of that may be the need to read numbers from a file. The file may contain an unknown amount lines, and each line an unknown amount of numbers separated by an unknown amount of whitespace or tabs. This may in some languages be a task demanding many lines of code. In Python it takes only two (example from "Python Scripting for Computational Science"[25]).

The code snippet in listing 2.1 solves the problem for us, and does it by calling highly optimized C code [25]. This code has been optimized by many people around the world, and it follows that it will be hard to beat the speed of the Python implementation [25]. This means that in the area of text processing, dynamically typed languages such as Python can be very efficient from both human and computer point of view [25]. Another benefit of Python is that it has very good options for moving CPU heavy operations into C, C++ or Fortran. This means that heavy computations on big arrays of numbers can be done in a faster language such C, while Python sets up all the needed variables and binds these number crunching loops together. Cython is one such tool, and will be discussed a bit more in section 4.2.3.

2.6 The framework

As with any new type of field of science, the research within genomic analysis is done with several techniques and assumptions. That makes it difficult to compare, reproduce and realize the full implications of various findings. The Hyperbrowser is a tool for analysing sequence-level genomic information built upon the Galaxy framework. Some of the more prominant tools are Galaxy, BioMart, EpiGRAPH and UCSC Cancer Genomics Browser. Biomart offers flexible export of user-defined tracks and regions. Galaxy offers a richer, text-centric suite and operations. EpiGraph has a solid set of statistical routines focused on analysis of user-defined case-control regions. While UCSC Cancer Genomics Browser visualizes clinical omics data, as well as providing patient centric statistical analyses [36]. Omics data are fields in biology ending in -omics, such as genomics and metabolomics.

2.6.1 Galaxy

Thanks to new sequence technology, there exists a vast amount of sequenced genomic data, but for most biologist, there is a void between accessing all this data and translating it into biological knowledge. The main problem might be the immense size of the genomic data sets [5]. Another problem is the incompatibility between formats. This is often solved by small custom one-off scripts, and while these scripts might be simple, they are a real problem to reproducibility if not made publicly available. If the scripts are made available, they often have confusing or command line only interface [5]. Rarely does the output of one tool work directly with another tool. This means it is practical for for biologists to have tool to use on hetrogenous data sources; Galaxy offers datatype converters for such data sources.

2.6.2 Hyperbrowser

The Hyperbrowser is a tool for comparative analysis of sequence-level genomic data by mixing genomics, computational science and statistics. It is based on an abstract representation of generic genomic elements as mathematical objects. Where hypotheses of interest are translated into mathematical relations. By using mathematical models, one can create randomized tracks with track structure preservation to build problem-specific null models about relations between tracks [36]. Some analyses of potential interest for two tracks, T1 and T2, may be: are the T1 points overlapping more with T2 than expected by chance, does the midpoints of on track overlap more than expected [36]? Information reduction of a track, from segments to points, may open for several new analyses, and this is handled dynamically within the system. analyses may be done globally or locally. A global analysis is an investigation of certain relation between two tracks as a whole, while local analysis is based on dividing the tracks into smaller units, called bins, and then performing the analysis on each bin separately [36]. Local analyses may be of interest to check if there may be any relation between certain biological mechanisms [36]. The local analyses can be used to see where there are any relations between a trait track and different cell types.

The Genomic HyperBrowser is open source and based on the Galaxy code. It is implemented in Python. One of the weaknesses of pure Python is slow runtime, see section 2.5.1. Therefore the HyperBrowser is built in two layers. On top it is created by Python for flexibility, and at the lower level all the tracks are stored as low-level vectors, numpy.ndarray [36]. This gives efficient storage, and allows the use of highly efficient numpy operations to ensure speed [36]. The system uses a web-based interface with an easy-to-use entry point. However, all the different analyses and how they interact with different tracks, and all the different null models, can make it difficult to get an overview and not so easy to use. To help with this a step-by-step approach has been used so that only the relevant and possible options are available at each stage. When something is selected, the HyperBrowser

tries to do short runs on all the implemented analyses, using a small part of the selected data, if the analysis fails, it is removed from the list of possible analyses [36].

2.6.3 Batch line commands

The HyperBrowser has the option to run batchline commands. That is a way of automatically running many tests on many tracks with just a few lines of code. This also makes for easier reproduction of results, as the batchline commands can be included in any report and rerun to do identical tests again. Variables can be defined as constants, or dynamically, so that is changes for each run.

```
1 @TNbins=<Path to disease>
2 @TN2s=Private:GK:Psych:DHSs:*
3 @binning=track|@TNbins
4 hg19|@binning|@TN2s||NumBinsContainingElementsStat(tf1=↩
      TrivialFormatConverter)
```

Listing 2.2: Example of batch lines

Listing 2.2 is an example of how a run could be written with batchlines. The @ is used to define variables, such as the trait track, and what tracks that trait is to be compared against. In this example, the variable @TN2s will represent all subtypes of DHS. Such as A549, CD19 and HAEpiC. The final line represents the actual command run. A list of processed command lines will be generated, such that the variables have been swapped with what they represent. In this, case that means it would be a list of batchlines, where there is one batchline for each subtype of DHS. The * is what tells the HyperBrowser that there may be more than one subtype to use.

2.7 Legacy Code

A definition of legacy code may be code gotten from someone else. It is someone else's code [16]. Some may associate legacy code with tangled unintelligible structure, code that you have to work with but do not fully understand [16]. In industry it is often associated with difficult-to-change code, code that is hard to fully understand [16].

There are mainly 4 reasons to change legacy code, adding a feature, fixing a bug, improving design or optimizing resource usage [16]. Adding a feature may include many things, it may be changing something while adding, it might even be removing something to add something else. Or it may be to add something completely new outside the old code. The HyperBrowser may be seen upon as legacy code, it is not very well documented and it seems like every part of it is dependent on some other part. There are mainly two ways to change something in a system of legacy code, one is the *Edit and Pray* and the other is *Cover and Modify* [16]. *Edit and Pray* is more or less the standard in industry [16]. With *Edit and Pray*, it is usual to build up an understanding of the code, then plan the

changes, then carefully make the changes. When the changes are done, the system is run to sees if it works. If it does, the next step is to prod around looking for mistakes, and to check if it all works as it should. The prodding is really important as it is done to make sure that nothing broke during the implementation. *Cover and Modify* uses a slightly different approach. In this method it is usual to cover the code with tests before any changes. That way the test will immediately fail if the changes break any dependencies or any input or output changes. By covering the code with tests before the change it is very easy to see what the changes result in, and if it has any unintended side effects.

2.7.1 Unit testing

The definition varies, but it refers to testing the smallest components in the code. In procedural (similar to functional programming) code it is usually methods or functions, while in object-oriented code it is usually classes or objects [16]. Testing these parts in isolation can be difficult, because functions usually use functions that use functions, and classes are usually dependent on other classes. The isolation part is very important. If the test is large, some problems arise, such as where is the problem located, and it takes too long time and how to make sure all possibilities are covered. These problems are easier solved in unit tests than in higher level testing. Unit tests make it possible to run pieces of code independently, or they can be grouped together and run under the same conditions. Unit tests also make it easy to localize problems. Two qualities of good unit tests are that they run fast and they help localizing problems [16].

A unit test that takes more than 1/10th of a second is slow. Unit tests are supposed to be small and local. If you start to group the unit tests together, it takes much longer. Say you have 1000 methods, and about 10 tests for each. That would make it 10000 tests, and if each test take 1/10th of a second, that makes it 1000 seconds, or 16 minutes. That is not fast enough for unit tests, the feedback should be almost instant. Otherwise it is very easy to find excuses not to run the tests. If a test uses a database, communicates across a network, touches a file system or need to change anything in the environment (like configuration files), it should not be considered a unit test [16].

It is usual to have to change some code to be able to add tests to untested legacy code. And that gives rise to *The Legacy Code Dilemma* as stated in "Working Effectively with Legacy Code" [16].

> When we change code, we should have tests in place. To put tests in place, we often have to change code.

One way to work around this dilemma is to use the legacy code change algorithm [16]. Firstly one find the change points, the points where the code needs to be changed. When these points are found, one can find the possible test points around that. Then one breaks the dependencies and write tests. Then all left is to make the changes and refactor the code [16].

2.7.2 Adding a feature

One of the most powerful ways of adding a feature to any code is the use of test-driven development (tdd) [16]. The idea is to write a test that will fail in the beginning, then you develop the code to make it pass. The benefit from this is that you need to clarify the input and output of the method before any code is written. The steps for tdd are as described in "Working effectively with Legacy Code"[16]:

1. Write a failing test case
2. Get it to compile
3. Make it pass
4. Remove duplication
5. Repeat

Following is an example of creating a division method in Python. This method will be divide10byX(x), and it will divide 10 by the user given x.

Writing a failing test case

The fist thing to do is to create a test that calls the method to be added, with values such that it is easy to control the result. This test case will immediately fail since the method is not yet created. For the same reason, this code would not have compiled if it was a compiling language.

```
1 def testDivide10byX()
2     res = divide10byX(2)
3     assert(res==5)
```

Listing 2.3: Legacy Code: Failing test case

Get it to compile

If this was a compiling language, the next step would have been to make it compile. That is done by creating the wanted method, but make it return something useless. Say for example the boolean False. This would make the code compile, but since this is Python is only changes the error message. The assertion would still fail when running the code, this due to the method not returning what is expected.

```
1 def divide10byX(x):
2     return False
```

Listing 2.4: Legacy Code: Making it compile

Make it pass

Next up is to make the test pass. This is done by making the method do as originally planned. In this case it is a matter of returning 10 divided by x. Now the test will tell if the method is implemented correct so far.

```
1 def divide10byX(x):
2     return 10./x
```

Listing 2.5: Legacy Code: Make it pass

Remove duplication

In this part of the algorithm one try to factor out parts of the code that do the same, and whenever that is found, factor it out into another method. This is done to make the code easier to maintain and read. In this example there is no duplication, but an example could be if another method existed, that divides 20 by x. Then it might be natural to create a method that can divide any number with any number, and then make the other two methods call that one, where one sends in 20 and x, and the other sends in 10 and x.

Start over

The final step step is to start over again, find possible points of errors and make tests for them. Possible errors for this example are integer division, and division by zero. By adding tests for possible errors, it is shown that these cases are accounted for and tested. It also helps others understand the code, and makes it easier to avoid breaking dependencies.

2.8 Essential articles

Here follows some summaries of some articles that are important for the design of this thesis. They have been important in selecting what genetic regions to include and also what type of analyses to include. Many of the technical terms used are explained in table 2.1. The articles summarized are “Genomic Regions Associated with Multiple Sclerosis Are Active in B Cells” [13], “An integrated encyclopedia of DNA elements in the human genome” [15], “Systematic Localization of Common Disease-Associated Variation in Regulatory DNA” [29], “Chromatin marks identify critical cell types for fine mapping complex trait variants” [39], “Predicting Cell Types and Genetic Variations Contributing to Disease by Combining GWAS and Epigenetic Data” cite[18] and “Next-generation sequencing in understanding complex neurological disease” [20]. Unless otherwise stated, all the facts in the summary will be from the article summarized.

Cell type	Explanation
B cells	A type of white blood cells [3]
Hepatocytes	A certain type of liver cell [34]
Fibroblasts	A cell found in connective tissue [17]
Keratinocytes	Cell in outer layer of skin [24]
CD-20	A protein primarily in the surface of B cells [38]
Rituximab	An anti-CD20 monoclonal antibody, destroying B cells [13]
Linkage disequilibrium	A non random association [21]
de novo	Latin: From the beginning [12]
Rheumatoid arthritis	Chronic inflammation of the joints [35]
LDL	Low-density lipoprotein, generally referred to as bad cholesterol [9]
cDNA	Complementary DNA made from mRNA [11]
GWAS integrator	Tool allowing cross database search [43]
HaploReg	Tool for exploring annotations on the non-coding genome [41]

Table 2.1: Explanations of terms and names used in essential articles

2.8.1 Genomic Regions Associated with Multiple Sclerosis Are Active in B Cells

So far, more than 50 genomic regions have been shown to influence the risk of multiple sclerosis (MS), but it is still largely unknown in which cell types these variants acts. Hence, it is hypothesized that these regions would co-localize with regions known to be active in B cells, due to the role B cells have been found to have in MS.

Background

Multiple sclerosis (MS) is a disorder affected by both genetic and environmental factors, and it damages the central nervous system. In one research they found evidence of 57 genomic regions being associated with MS, but the knowledge of how those regions are involved in MS development is limited. Rituximab's (explained in table 2.1) success in treating MS have heightened the interest in B cells' role in MS, and now other anti-CD20 monoclonal antibodies are undergoing clinical tries.

Many of the SNPs associated with MS do not lie in coding regions of the DNA, which makes it likely that they affect the disease risk through a gene regulatory role. The lack of distinguishing sequence signature makes those regulatory elements in the genome much harder to identify than protein-coding genes, although this does not mean they are less important. To find these regions, one can use chromatin profiling, which is a powerful tool to detect regulatory activity. The chromatin region of a cell does among other roles, determine which regions of the genome are accessible to the binding transcription factors and whether transcription occurs or is repressed. It is

also distinctive for a certain cell type. A study has mapped the chromatin marks of nine different cell types to characterize regulatory elements and their cell type specificities. Among the cell types were B cells, hepatocytes, fibroblasts and keratinocytes. See table 2.1 for explanations. Since many of the SNPs associated with MS might influence disease risk through a gene regulatory role, it is plausible that these SNPs should be observed with a higher rate in the regulatory regions for the cell types causing or affecting MS. Hence, they hypothesize that they should find a bigger overlap between MS SNPs in the regulatory regions in B cells compared to non-immunological cell types.

Methods

The genetic variants for MS were collected from research done by the International Multiple Sclerosis Genetics Consortium (IMSGC) and the Welcome Trust Case Control Consortium 2 (WTCCC2). Information about the different cell types was collected from the ENCODE project (more information in section 2.8.2). The Genomic HyperBrowser was used to do all analyses. The enrichment was calculated as the ratio between the proportion of a state covered by MS, and the proportion of everything except the chromatin state covered by MS. In other word, they compared the overlap between MS and the chromatin state with the overlap of MS and regions outside that chromatin state. To find out if the overlap was higher than expected by chance, they created a null model where the location of individual chromatin intervals varied randomly and the MS regions were fixed. Then they compared the real data overlap with 20000 Monte Carlo samples of the null model. The chromatin states they looked at were active promoter (AP), weak promoter (WP), poised promoter (PP), strong enhancer (SE), weak enhancer (WE), polycomb repressed (PR), heterochromatic (H), insulator (I), strong transcribed (ST), weakly transcribed (WT) an repetitive/CNV (Rep/CNV).

Results

On a global scale the enrichment values varied a lot, from 0.34 (lowest) in H, to 3.07 (highest) in SE. But the overlap between MS and AP, WP, PP, SE and ST was statistically significant. The chromatin states with the significant overlap on global scale were also the ones with the highest amount of significant bins, where the bins are chromosome arms. High overlap between MS and active chromatin states in B cells is not necessary an indication of any relation between the two. Therefore tests was also done with hepatocytes, fibroblasts and keratinocytes, which are thought not to be relevant to MS. This showed that in some chromatin states, MS overlapped more than by chance in all cell types. But the number of significant bins and the enrichment value tended to be higher in B cells. In AP, SE, WE and ST regions the overlap between MS and B cells was significantly higher than the other cell types.

2.8.2 An integrated encyclopedia of DNA elements in the human genome

The Encyclopedia of DNA Elements (ENCODE) project was started in 2003 and aims to describe all functional elements encoded in the human genome. Some of the elements mapped are RNA transcribed regions, protein-coding regions, transcription-factor-binding sites, chromatin structure and DNA methylation site. After mapping these regions, they found that 80.4% of the genome was covered by at least one ENCODE element. RNA covers the biggest part of the genome.

Transcription-factor-binding sites provides a way to explore chromatin properties. Transcription factors often have more than one function, it can bind variations of genes and DNA sequences and different patterns of chromatin marks.

The database with GWAS SNPs is growing fast, but 88% of associated SNPs are found in noncoding DNA. They compared 4860 SNPs believed to be phenotype specific, with 4492 SNPs organized in the National Human Genome Research Institute (NHGRI) GWAS catalogue. The result was that 12% of SNPs overlap with transcription-factor-occupied regions, and that 34% overlap DHS regions.

The new elements found in this research show a statistical link with sequences connected to diseases in humans, thus these elements can help understanding of the diseases.

2.8.3 Systematic Localization of Common Disease-Associated Variation in Regulatory DNA

The majority (93%) of disease- and trait associated variants emerging from GWAS studies lie within noncoding sequences. It was defined an average of 198180 DHSs per cell type, then 5654 noncoding genome-wide significant associations (5134 unique SNPs) was looked at. These SNPs covered for 207 diseases and 447 quantitative traits.

This gave an enrichment of 40% GWAS SNPs in DHSs, where 76.6% of all noncoding GWAS SNPs either lie within a DHS (57.1%, 2931 SNPs) or are in complete linkage disequilibrium (LD) with SNPs in a nearby DHS (19.5%, 999 SNPs). To confirm these numbers they used variants from the 1000 Genomes Project on individuals of European ancestry. Here they found significant enrichment for both SNPs within DHSs, including for variants in complete LD with SNPs in DHSs.

GWAS SNPs was the classified into three groups, non replicated, internally replicated (confirmed in a second population in the initial study) and externally replicated (replicated my an independent study). This showed that externally replicated SNPs had higher enrichment in DHS than the internally replicated SNPs, which had higher enrichment than the non replicated SNPs. Almost 70% of the externally replicated SNPs was found in DHSs. This shows that many SNPs are functional and that unreplicated and weaker associations may hide the true enrichment between SNPs and DHSs.

Many disorders have been linked with things happening in early gestational stage, and hence the DHS regions with GWAS variants were checked for activity during fetal stages. 88.1% of the noncoding SNPs was found to lie within DHSs active in fetal cells and tissues, and 57.8% of DHSs containing those SNPs are active in both fetal and adult cells. While 30.3% are only in the fetal stage.

2.8.4 Chromatin marks identify critical cell types for fine mapping complex trait variants

The idea is that if trait-associated variants change regulatory regions, they should fall within chromatin marks in relevant cells. Therefore 15 chromatin marks were examined, and that showed that those highlighting active gene regulation were phenotypically cell type specific.

It was assumed that variants close to or directly under tall chromatin mark peaks in specific cell types might be involved in cell type specific gene regulation, and that variants far from chromatin mark peaks are much less likely to have a direct role in gene regulation.

Only traits with at least 15 reported associations in European populations were selected, and cropped by LD. This gave a set of 510 independent SNPs associated with 31 diseases. Defining the genomic locations and heights of 15 chromatin marks in 14 cell types, gave 4 chromatin marks with significant phenotypic cell type specificity. The significant ones were H3K4me3 and H3K9ac, which both are known to highlight active gene promoters. To check the reproducibility, 6 different chromatin marks in 38 different cell types was collected from the Epigenomics project. Again the H3K4me3 was the most informative mark. The H3K9ac was much less significant, which may be due to fewer cell types analysed. Another test were done, where they shifted the chromatin peaks locations. This resulted in a lowered significance and hence it suggests that it is phenotypically association and not close proximity to gene structures that gives the score.

Chromatin was divided into two groups, those falling within nearby promoter regions, and those falling outside of promoter regions. This gave an enrichment of H3K4me3 in strong and disease associated enhancers when redoing the analyses, although H3K4me3 markers are generally not thought of as being enriched in enhancers.

37 SNPs associated with LDL concentration was tested for overlap with H3K4me3 marks in different cell types to confirm the results. The results showed that the 37 SNPs implicated a total of 1501 H3K4me3 peaks in 34 different cell types, where the most significant cell type was adult liver tissue. This corresponds with the idea that these variants should imply regulatory activity within the liver.

31 SNPS associated with rheumatoid arthritis was also tested against H3K4me3 marks. This found 1328 peaks in 34 cell types, with the most significant association in CD4+ T cells, in particular CD4+ regulatory T (T_{reg}) cells. The phenotypically similar CD4+ memeory cells were also highly significant. Trying the same for newer SNPs gave an increased significance of the enrichment for CD4+ T_{reg} cells.

In certain cases it is possible that several cell types could be implicated in a disease. When looking at 67 SNPs for type 2 diabetes, it was found a total of 2776 H3K4me3 peaks within 34 different cell types. With the most significant being pancreatic islets and the liver. Combining these to cell types gave an increased significance, even more significant than the cell types individually. That combination was also more significant than all other possibilities of paired cell types. Removing one of them, increased the significance for the other. Both liver and islet cells are known to have a key role in mediating glucose synthesis, insulin secretion and diabetes.

Discussion

It was found that chromatin marks highlighting regulatory regions, such as H3K4me3, H3K9ac and DHSs, overlap with phenotypically associated variants, and that this overlap is phenotypical cell type specific. It also supports the hypothesis that many complex disease and trait alleles might act by influencing gene regulation in a cell type specific manner. The approach used is sensitive to the diversity and number of cell types analysed, but it might still be used to connect phenotypes to specific cell types or mapping phenotype associated SNPs to potential regulatory variants.

2.8.5 Predicting Cell Types and Genetic Variations Contributing to Disease by Combining GWAS and Epigenetic Data

Background

"Asthma is a chronic inflammatory disease, characterized by reversible airway obstruction and increased bronchial hyperresponsiveness" [18]. There have been many GWAS studies on asthma due to the estimate that about 35%-80% of the variation in the risk of asthma is genetic variation based. These studies have found a large set of SNPs significantly more frequent in asthma patients than in the healthy control group. But linking the asthma SNPs and development of disease is not simple, this is because the majority of the SNPs identified are in non-coding regions. And hence they have no expected phenotype or function. Location of enhancers such as H3K4me1 and H3K27ac has been shown to be highly cell type specific, combining that with the knowledge that disease-associated SNPs are enriched in enhancers, may be used to connect cell types to diseases. This would be done by finding what enhancers the SNPs are enriched in.

Results

All known asthma SNPs were collected from the GWAS integrator database, giving a total of 131 SNPs. Based on HaploReg the tight genetic linkage to these SNP was also collected. As a control set, SNPs not associated with asthma were used. It was found that the asthma SNPs are highly enriched

in coding sequences compared to the control SNPs. However the biggest amount of asthma SNPs are in introns and intergenic regions, non-coding regions known to hold enhancers. It was also found that in CD4+ T cells there is a significant enrichment of asthma SNPs in promoter and enhancer associated regions. It was shown that the genomic location of enhancers marked by H3K4me1 was primarily determined by cell type.

All enhancers were collected into groups, so that there was one group for each cell type, ADM stem cells, liver cells, kidney cells, adipose nuclei, skeletal muscle cells, brain cells, breast cells and CD4+ cells. For these groups it was determined how many of the asthma-associated SNPs were found in enhancers, and calculated the enrichment. The biggest enrichment (2.11) was found in CD4+ T cells, there was also found enrichment in liver, adipose nuclei and adipose stem cells. While there was a very low enrichment in skeletal muscle, breast myoepithelial and brain cells. This makes sense as these cell types are believed not to be of importance to asthma. Of the 884 SNPs that fell into enhancers in one or more of the eight tissues, 443 were found in only one cell type while 39 were found in all eight cell types. By looking at the enrichment of asthma SNPs in enhancers in CD4+ T cells compared to how many cell types has this enhancer, one could see that the fewer cell types, the higher the enrichment. Highest enrichment was therefore found in enhancers unique to CD4+ T cells, while the enrichment in skeletal muscle cells decreased with stricter uniqueness to enhancer. It was also found that asthma associated SNPs located in enhancers overlap with Transcription Factor Binding Sites (TFBS) four times more often than those not located in enhancers. This means that disease SNPs in non-coding regions are more likely to be functional, and may do so by disrupting binding of transcription factors.

Discussion

It is difficult to predict how genetic variations affect disease development, as the effect may be limited to a certain cell or effected by environmental factors. To help void these problems, the techniques from this article may be used. The authors suggest that this method may be even more powerful if all SNPs were reanalysed, and then limiting the method to the SNPs active in the genomic regions of the cells of interest.

2.8.6 Next-generation sequencing in understanding complex neurological disease

Techniques

The first complete read of the human genome was obtained in 2003, and since then there has been an explosion of people who have had their genome fully or partially sequenced. Only recently, though, has it been possible to use it on complex diseases. Complex diseases are defined as diseases where multiple factors affect the risk factor such as multiple

genetic and environmental factors, and where these factors will affect individuals differently. GWAS research has been extremely efficient in finding genetic risk loci, but very few of these have a direct detectable effect on gene expression. They do more commonly have other regulatory effects, such as influencing gene expression. For example, 47.6% of some SNPs are predicted to alter transcription factor binding, while only 7.3% are shown to alter gene expression. For disease associated SNPs there has not yet been found a function, and that is believed to imply that the true casual variant is likely to be a nearby variant.

The usual way to use next-generation sequencing is to align multiple, short, overlapping reads of fragmented DNA against a reference genome. If no such reference genome exists, it can be assembled “de novo”. Another method of sequencing is whole-exome sequencing. This uses the same method, but complementary strands to known exons are used to extract fragments covering exonic regions prior to sequencing. This makes the analysis cheaper and less computationally demanding because it requires less sequencing. A drawback is that it will only capture variants inside exons, and hence lose any causative variants elsewhere. This of course is not optimal as it has been shown that the majority of variants are located outside the exons. Many causative variants for complex neurological disease may be found by next-generation sequencing, but because of the high number of variants the chance for false positives is also high. For example, whole-exome sequencing on MS patients did find more than 58000 variants in each individual. This shows the need for integrating information and knowledge from GWAS to help select plausible candidates. As more genomes get sequenced, it will become easier to understand how complex neurological diseases works.

Next-generation sequencing can also be used for sequencing mRNA. This is done by using probes for polyadenylated mRNA and reverse transcriptase to generate cDNA, which then can be sequenced using conventional techniques. It is very important that this type of sequencing is done on the correct cell types, as there is evidence that genetic effects on expressions may be cell type dependent. So far the use of such sequencing has been very limiting, but in the future it may become easier as more information of the normal human brain tissue is mapped. This is done by collecting samples port mortem from different development stages in humans. The main difficulty is to collect tissue, since that means autopsy samples must be studied, and that is generally a poor replacement for living tissue, specially since RNA very easily degrades fast when dead.

Another use of next-generation sequencing is chromatin immunoprecipitation. In this method, a protein of interest bound to fragments of DNA is separated using antibodies. The DNA fragments can then be sequenced and mapped back to reference genome. This provides a map of where that protein is binding, and hence provides data on functional genomics. It has been shown that a significant amount of GWAS associated SNPs lie in transcription binding sites. So this method can give an idea on how genetic variants effects disease development. A downside to this method is that it needs a strong theory that can determine the target of the antibody used.

"Epigenetics is the study of reversible modifications to the structure of DNA associated with gene expression differences." [20] Some think that epigenetic variations may be the cause of several complex diseases. Next-generation techniques can be used to search for regions of the genome enriched for modified forms of chromatin. Epigenetics is believed to be cell and time dependent, and hence selecting correct tissue and timing is important.

Use

Identifying causative variants in previously undiagnosed or complex diseases has already been done, and hence it might be said that the most direct use is within molecular diagnostics. Next-generation sequencing is also likely to have an important role in diagnosis of complex phenotypes. Another use is risk prediction and genetic counseling, where the popularity of direct to consumer (dtc) shows the public's interest in that. Companies like 23AndMe and Navigenics are selling self tests, that then are analysed and profiled for common diseases through SNP genotyping. See section 2.4.3 for more information on this subject. So far, personalized medicine has yet to bear fruits, but next-generation sequencing may in the future allow for treatments to be tailored to individual patients in a way that previous genetic techniques could not, as noted in section 2.4.

Next-generation sequencing techniques have produced huge amounts of data. This demands for efficient statistical methods, that will allow for quicker and more powerful analysis of the data.

Part II

Method and implementation

Chapter 3

Method

In this chapter the theory behind the tool and algorithm will be presented. The first section is about how it is possible to store and think of genetic data as mathematical segments over the natural numbers, and how that makes it possible to do analyses based on mathematical models. The final section will cover the analyses used by the tool in the HyperBrowser.

3.1 Genomic Tracks

There are many different ways to represent genomic information, and as with most new fields, there is no general consensus on how this should be done. In bioinformatics, standards such as ensemble, refseq and uniprot exist side by side. This can make it hard to compare and duplicate results and tests. In ensemble there may be several names for the same gene at almost the same position, if not the exact same position. To be able to work with the different standards it may therefore be necessary to use translation programs or compare positions to find the equivalents, but there is no guarantee that such an equivalent exists. To avoid this, the HyperBrowser uses tracks as described in this section. That way, one only have to compare positions. It stores chromosome, position and the alleles. Figure 3.1 shows how some genomic tracks can be visualized. Figure 3.1a shows genetic points, a point might be the position of a single allele mutation, or other points of interest. Figure 3.1b shows representation of genetic segments. A segment can for example be a exon or intron. Each segment represents a specific place on the track. Figure 3.1c represents the same as fig. 3.1a but this time each point is given a weight or value. One of the uses for this may be if several tracks are merged together, then each point can be valued according to how many of the tracks have a point in that position. Figure 3.1d shows weighted or valued segments. The use is about the same as fig. 3.1c. To store the genomic tracks, it is only necessary to keep the relevant info. That is the boxes or thin vertical lines in fig. 3.1. When HyperBrowser processes a track, it creates a track object containing information about that track. From such a track object it is possible to collect all the segments and points as arrays. That is, for each chromosome it will return one array containing the start positions and one

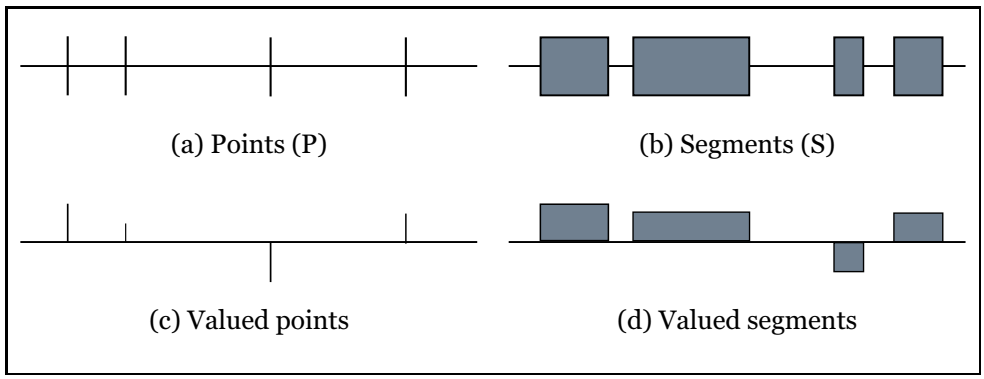


Figure 3.1: Different ways of visualizing track types. Figure 3.1a may be a genetic point, like a SNP. Figure 3.1b may be a SNP with flanks, creating a segment of DNA. Figure 3.1c may be point given a certain value based on some test. Figure 3.1d may be a segment given a certain value based on some test.

array containing the stop positions. These arrays are linked in such a way that the start position at index i in the start array, belongs to the stop position at index i in the stop array. The start array is always sorted, and hence the equality follows like this:

$$\text{startsAsArray}[i] \leq \text{startsAsArray}[i+1] \quad \text{if overlap allowed} \tag{3.1}$$
$$\text{startsAsArray}[i] < \text{startsAsArray}[i+1] \quad \text{if overlap not allowed} \tag{3.2}$$

The stop array can not be assumed to be sorted, since the stop position at index i is linked to the start position at index i. The exception is if no overlap is allowed, then one can assume it to be sorted as well. If overlap is allowed, the stop position at index $i+1$ may very well be smaller than the one at index i. The positions in the arrays can de seen upon as positions along the natural number line, and hence one can start to look at this a bit mathematically. The ideas to the definitions used are based on the work done in [19], but not identical. It has been modified and simplified to what is needed for this thesis.

The definition of a metric space d on the natural numbers is as follows, given $\forall x, y, z \in \mathbb{N}$ the following holds:

$$1 : d(x, y) \geq 0 \tag{3.3}$$
$$2 : d(x, y) = 0 \iff x = y \tag{3.4}$$
$$3 : d(x, y) = d(y, x) \tag{3.5}$$
$$4 : d(x, z) \leq d(x, y) + d(y, z) \tag{3.6}$$

A metric space on the natural numbers is the regular distance function, which works well for calculating the size of a segment or a point:

$$d(a, b) = |a - b|, \quad a, b \in \mathbb{N} \tag{3.7}$$

This gives the size of a segment going from a to b, where b not inclusive. A point is when $b = a + 1$. This means that a segment will consist of a subset S of natural numbers, which can be defined as:

$$S(a, b) = \{s \in \mathbb{N} \mid a \leq s < b\} \tag{3.8}$$

And we define a point to be:

$$P(a) = S(a, a + 1) \tag{3.9}$$

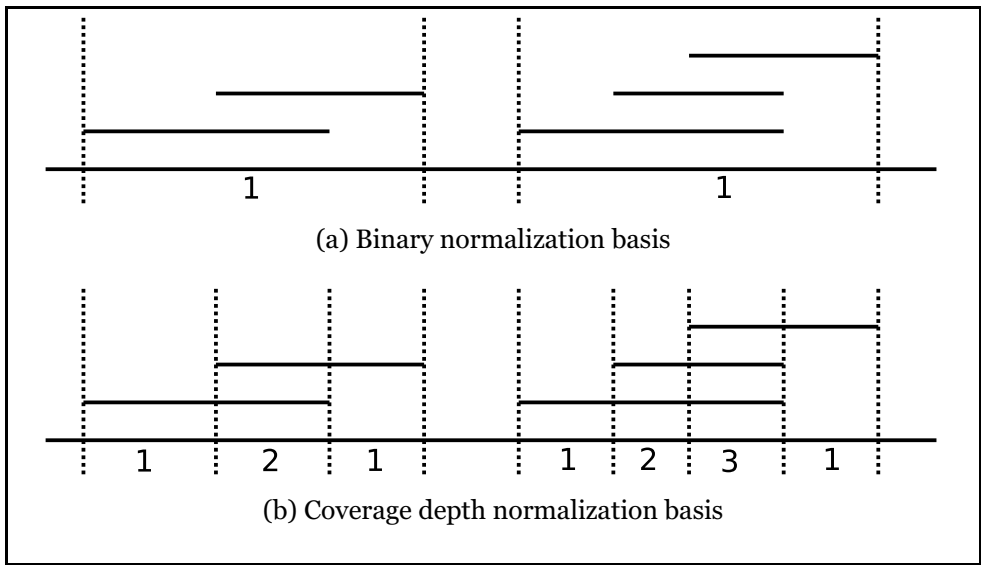
By defining it like this we can say that a segment consists of at least one point, and a segment is a point if and only if it has size one in the metric system. Take for example, a segment starting at 1 and ending at 3, inclusive. That gives $S(1,4) = \{1,2,3\}$ and $d(1,4) = |1 - 4| = 3$. A point is then $P(2) = S(2,3) = \{2\}$ and $d(2,3) = |2 - 3| = 1$. This shows that a point can be within a segment; it also allows for overlapping segments. Another practicality is that it allows tracks to contain both segments and points, as long as it is represented as segments. This fits well with how one can collect the tracks from the HyperBrowser, as arrays of integers, and this metric space is the basis which the overlap algorithm described in section 4.2.3 is built around.

By representing these tracks mathematically it is also easy to generate random reference tracks. The reference track can be used to say whether the similarities are greater than they would be by chance. The randomized track can be generated fully randomized, or it can be generated according to biological rules to make it more realistic.

3.2 The analyses

There are mainly five analyses that can be selected in the tool, and used for different statistical purposes. The following sub-sections are meant to give an overview of what kind of tests the different analyses does. These analyses were already implemented inside the HyperBrowser, but they could only be selected individually before. For simplicity T1 and T2 will represent two arbitrary tracks. They may very well be looked upon as trait track (T1) and a random subtype of a genetic region (T2). Each of the analyses below will be done for each chromosome in each of the tracks separately. Hence it does not work on the full track as a whole, but it takes chromosome one in T1 against chromosome one in T2, then chromosome two against chromosome two, and so on.

There is a option for setting a global p-value threshold when selecting a hypothesis testing as analysis. The tests will run until a p-value is found that is better than the threshold, or until one of the other criteria set in the code is met. The other criteria are, 20000 test runs, or 10 test samples with better results than the real case.



(a) Binary normalization basis

(b) Coverage depth normalization basis

Figure 3.2: Shows how the different normalizations are calculated based on overlap between segments

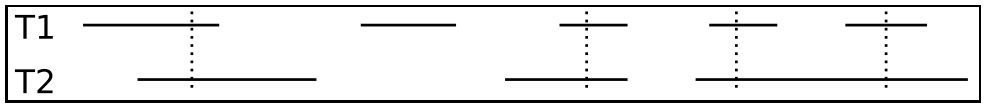


Figure 3.3: Shows how regions touched between T1 and T2 is calculated, 4 segments of T1 touches a segment in T2.

3.2.1 Binary and coverage depth normalization

The HyperBrowser uses two different different ways of calculating the value of a certain position on a track. In this case one can imagine a track T3 which contains T2 and all the other subtypes for that region, where the tracks in T3 are allowed to have overlapping segments. T3 then becomes the normalization track that is used to say something about the size of T2 compared to all the other subtypes. If T2 is large in T3, then one should expect that T1 overlaps more with T2 than with the other tracks in T3 if it was random. Thus the T3 is used to find out if T1 overlaps more with T2 than size accounts for.

Binary and coverage depth are two different ways of defining the specifics of T3. With binary T3 flattened down, it ignores how many segments might be overlapping and just stores if there at least one segment there or not. Figure 3.2a visualizes how the binary track is calculated.

Coverage depth values each point according to how many segments was overlapping there. Hence if T3 has 4 segments overlapping at position 1000, that position is given value 4. Figure 3.2b visualizes how coverage depth is calculated.

3.2.2 Regions touched

Regions touched can be calculated in two ways. It can be done by looking at the entire segment, or it can be done by looking at the midpoints. If regions is selected it will count how many segments in T1 touch a segment or point in T2. If the segment in T1 touches anything in T2 it is counted as one, and it moves to the next segment in T1. This means that if T1 contains 40 segments, regions touched on a global scale will be between 0 and 40 inclusive. Figure 3.3 shows how 4 segments in T1 touch segments in T2. The midpoints in the segments in T1 will be be found if midpoints is selected, and the number of midpoints touching any segments will be counted.

This analysis also allows for hypothesis testing. It then generates a large amount of random segments or points, depending on what is chosen of midpoints or regions. The result is presented as a p-value, saying whether the result is higher than what can be expected.

3.2.3 Enrichment

Enrichment says something about the relation between how much of T1 is inside T2 compared to outside T2. The mathematics behind it is best explained by looking at the actual calculation. For simplicity we define tp, fp, fn and tn as:

- tp is the intersection of T1 and T2
- fp is how much of T1 that does not cover T2
- fn is how much of T2 that does not cover T1
- tn is how much of the whole genome that does not cover any of T1 or T2

This creates a set of variables that makes it possible to say something about the relations between these two tracks.

$$1inside2 = \frac{tp}{fn + tp} \tag{3.10}$$

$$1outside2 = \frac{fp}{fp + tn} \tag{3.11}$$

$$enrichment = \frac{1inside2}{1outside2} \tag{3.12}$$

In this case the result is T1's enrichment in T2. For enrichment, one can choose to use coverage depth or binary normalization as basis for T2.

3.2.4 Hypothesis Testing

Hypothesis testing checks if the enrichment of T1 in T2 is larger than expected. This is decided by randomly generating tracks, and then

comparing it with the original result. This type of testing takes many statistical variables, but for this tool they have been preset in the code and hence hidden from the user. What the user can choose is whether it should be normalized based on binary or coverage depth.

3.2.5 Count Points

This analysis counts the number separate segments in T2. This means that if two segments overlap, they will be merged and counted as one.

3.2.6 Average Segment Length

This analysis calculates the average segment length in T2.

Chapter 4

Implementation

This chapter will cover the implementation of the tool and the new overlap algorithm. It will be explained how the tool works, and to some extent cover how it interacts with the HyperBrowser and how it uses the *GwasBatchLines* class. Then the *GwasBacthLines* class will be described in some detail, including the methods and how to expand it. The final section will be covering the new overlap algorithm, how it works and how it was compiled into efficient C code.

4.1 Genomic variation analysis tool

The tool is built in such a way that when it runs, it creates objects of a class named GwasBatchLines. Each genetic region will create a new object. The object keeps track of what type of analyses should be run with which settings, and it keeps track of what type of tracks belong to what analyses and regions. The next two sub-sections will explain the implementation of the tool and the method that creates the objects.

4.1.1 Inside the tool

Making selections

The tool is created as a part of the HyperBrowser, and hence it is web based with a graphical user interface. This also means that it inherits a lot of methods and structure from the HyperBrowser. Everything has to be created within a class, and that class needs some methods. One method is the *getToolName*, which returns the name of the tool as a string. Another method is the *getInputBoxNames*, this one returns a list of strings. Where each string is the name or text of the selection boxes in the tool.

Then comes the *getOptionsBoxY*(*prevChoices*), the Y represents the number of the box. And *prevChoices* is a collection of the previous selected options. It is this tool that creates the drop-down boxes and checkboxes. The drop-down boxes are created if the method returns a list of strings, where then each string becomes an option in the drop-down menu. The string selected from user is then placed in the *prevChoices* variable. This

implies that each of these methods have access to previously selected choices. The checkboxes are created when it returns an OrderedDict with tuples. A OrderedDict is a subclass of dictionaries in python. It supports the usual methods for dictionaries, but it also remembers the order the keys were first inserted [2]. Checkboxes can be grouped together, such that all genetic regions can be one group and the analyses in another group. One OrderedDict can contain many tuples, and each tuple becomes one checkbox. The tuple consists of one string, the text, and one boolean. If the boolean is preset as True, the box is pre-selected, and if it is False it is unselected. The string of the tuple is also the key to each box, and the key will return True or False depending on if the box is selected by the user or not.

The user should not have to think about things not relevant for the selection made. For example should the user not need to consider what type of normalization to use, if all he wants is the get the average segment length of some regions. This is solved by letting the methods return *None* if some condition is not met. The condition then being for example that an analysis needing normalization is selected before the user has to choose normalization. If the condition is met, it should return the menu like previously described.

The execute method

The execute method is called when the execute button is clicked. The execute method takes all the selections as input. For this tool the execute method analyses the path to the trait track, and collects information like track type and name. Then it uses that information together with the choices made by the user to set up the objects. This is done in a separate method, which takes the needed information as input. A new object is created for each genetic region to analyse, and each object is stored in a dictionary with the region as key. It then runs through the choices done by the user, and sets variables in the object to True according to what is selected. It returns the dictionary with the objects back to the execute method.

Then one and one object is sent to another method which collects all the subtypes for the genetic region in a list. For each subtype it collects all the batchlines from the object. It then inserts the subtype to each batch command and sends it to the HyperBrowser specific *runBatchLines*. This method returns a ResultDict for each batch command. A ResultDict is a HyperBrowser specific python dictionary with some extended functionality, such as a method to get the global result for all chromosomes.

It then collects all the global results, the result from the entire genome and not just in one chromosome, into a table. Where each row contains the result for one subtype, and the columns are the analyses. This table is written to a web page, by the use of methods writing tables in html. The *tableHeader* takes a list strings and generates the header line of the table. Each element in the list is the header for a new column. The *tableLine* creates the rows, and too takes a list of strings, where each string is written

to a new column. Here, each string becomes the html code for a link, pointing to a subpage with more detailed information about the results. By using the *getLink* method on the path to the subpage, one gets the html code for a link to that element. The *getLink* method takes in a string, which becomes the text for that link.

Another method uses the information from the table to draw the graphs using HyperBrowser methods again. First a path is created to a file, then the method *openRFigure* is used on that path. That allows the user to plot to the path using regular R commands. When the plot is done, one uses the file with *closeRFigure*. The plots collect the 20 subtypes with the highest enrichment and highest amount of regions touched and plots them in separate graphs. If there exists some p-values form hypothesis testing, this is plotted on top of the enrichment graph. If only hypothesis testing is done, then that is plotted alone.

4.1.2 Inside the GwasBatchLines class

GwasBatchLines is a class used to create an object for each genomic region that is to be analysed. It contains a lot of different boolean variables, saying something about what type of analyses are supposed to be ran on that cell type. Following will be an explanation on how that class works, and how it can be expanded with other genomic regions. So the following paragraphs are written to make it easier for future developers to understand how this class works. That will also make it easier to add genetic regions or to expand with more tests. For the impatient reader, the *init* method and the *getUnion* are the ones explaining how to add genetic regions, while the *init* method and the two sections about batchlines are needed for adding new analyses.

The init method

The init method can be seen in listing 4.1, and it is run when a new instance of the class is created. It accepts six variables, where the first is a string of a path to a trait track. It may be one stored in the HyperBrowser database, or it may uploaded and created by the user and hence be found in the history panel. There exists two different types of paths to tracks; one is to the database, and the other is a html link to the place the track is stored in history. They can look like this, where the first line is for HyperBrowser database, and the second line is the first part of a html link.

```
1 Private:GK:MS3dec12Base:GwasSnps:8 - Asthma_SNPs.txt
2 galaxy:bed:/usit/invitro/data/galaxy/galaxy-dist-hg-personal/database/...
```

Listing 4.2: Hyperbrowser track paths

It also takes in *trackName* as a string; this is the last part of the path to the trait track. This means that from the example above it would be *8 - Asthma_SNPs.txt* for the path to the HyperBrowser database. This is done to make it easier to present the results for that object later. The *traitTrack*

```
1  def __init__(self, traitTrack, trackName, NAME, trackType, version, ↩
       chromatinState = ''):
2      self.NAME = NAME
3      self.trackType = trackType
4      self.VERBOSE = True
5      self.touched = False
6      self.enrich = False
7      self.testing = False
8      self.observed = False
9      self.obsExp = False
10     self.mid = False
11     self.allreg = False
12     self.binary = False
13     self.coverage = False
14     self.countpoint = False
15     self.avgseglen = False
16     self.chromatinState = chromatinState
17     self.traitTrack = traitTrack
18     self.trackName = trackName
19     self.version = version
20     self.tf1Touched = ''
21     self.rawStat = 'ThreeTrackT2inT1vsT3inT1%sStat'%version
22
23     self.FT = '1.0'
24     self.GlobT = '1.0'
25     self.GT = '0.01'
26     self.numResamplings = '3'
27     self.mcParams = 'maxSamples=2000,numResamplings=%s,numSamplesPerChunk↩
           =25,'%self.numResamplings+\
28         'mThreshold=10,fdrCriterion=simultaneous,fdrThreshold=@FT,↩
               globalPvalThreshold=@GT'
29
30     self.DHSUnion = 'Private:GK:Psych:DHSs'
31     self.DHSUnionCoverageDepth = 'Private^GK^Psych^DHSCoverageTrack'
32
33     self.TFUnion = 'Private:GK:MS4:EncodeTfsExceptVdr'
34     self.TFUnionCoverageDepth = 'Private^GK^MS4^↩
           EncodeTfsExceptVdrCoverageFunction'
35
36     self.ChromatinUnion = 'Private:Anders:Chromatin State Segmentation:'
37     self.ChromatinUnionCoverageDepth = 'Private^GWAS^Chromatin^↩
           CoverageFunctionTracks^'
38
39     self.H3K4me3Union = 'Private:GK:Psych:H3K4me3'
40     self.H3K4me3UnionCoverageDepth = 'Private^GK^Psych^↩
           H3K4me3CoverageTrack'
41
42     self.enrichUnion, self.testUnion, self.testUnionFloat = ↩
           GwasBatchLines.getUnions(self, NAME)
```

Listing 4.1: Gwas Batchlines init method

is the actual path or link. The third variable is $NAME$, which tells what type of genetic region this analyses is to be run on. The name will be one of the genetic regions the user has selected run the analyses on, there will be a new object for each region selected. The variable $trackType$, is used later to tell the system what type of track the trait track is. If it is selected from the database, it is one type, while if it is selected from history it can be different types of tracks. From the example above, the track type for the history path would be *bed*. The version says something about which method will be used to calculate the coverage depth. It will either be $Version2$ or $Version3$, if it is $Version2$ it will use the old statistic, while $Version3$ will use the newer statistic explained in section 4.2. This variable changes the $rawStat$ string, and also affects which one of the merged region tracks that will be used. This option is kept in the code for analysing speed and correctness, but the choice is not included in the tool as $Version3$ is default. The final input is the $chromatineState$; this one is there to make sure the merged track for the correct chromatin state is returned.

The boolean variables are used to determine what batch lines are to be returned. Some will be set to true depending on selections made by the user in the tool, and then when the tool calls $generateBatchLines$ it will run through a list of if else tests to determine which commands to return.

The FT, $GlobT$, GT, $numResamplings$ and $mcParams$ are different variables used when running statistical analyses, and they are copied into the batch line further down in the code. They are extracted like this so that it is enough to modify it once for all different analyses. Line 30 to 40 are paths to different tracks used for the different genetic regions. They are sorted in groups based upon region. Hence the DHS are grouped together, and the same for TF, Chromatin and H3K4me3. The first one is pointing to all relevant subtypes of that region and at the same path is a track with all the subtypes merged together. This merged track can contain overlapping segments. The second track is the merged track, but without overlapping segments. How that track looks will be explained more in section 4.2.1. Another detail is that the : has been swapped with^. Some analyses in the HyperBrowser cannot use : as a divider, and because of this it it has been swapped with^.

The final line, line 42, sets up which of the merged tracks are to be used later based on genetic region selected and which statistic is used.

To add new genetic regions, they will have to be added like the previous ones: one which hold the path to the subtypes, and on containing the path to the merged track without overlap if it is to be included in testing the two algorithms.

To add more statistics, more boolean variables have to be added to the top, so they can be set be set from the tool.

The getUnion method

This method is there to make it easier to add more genetic regions. Its main purpose is to return the two correct paths, which is dependent on genetic region and version of the statistic. The method can be seen in listing 4.3. It

```
def getUnions(self, NAME, version):
    if NAME == 'DHS':
        if version == 'Version2':
            return self.DHSUnion, self.DHSUnionCoverageDepth
        else:
            return self.DHSUnion, self.DHSUnion.replace(':','^')
    elif NAME == 'TF':
        if version == 'Version2':
            return self.TFUnion, self.TFUnionCoverageDepth
        else:
            return self.TFUnion, self.TFUnion.replace(':','^'),
    elif NAME == 'Chromatin':
        union = self.ChromatinUnion + self.chromatinState
        unionCoverageDepth = self.ChromatinUnionCoverageDepth + self.↩
            chromatinState
        if version == 'Version2':
            if 'Active' in self.chromatinState:
                unionCoverageDepth = unionCoverageDepth + 'V2'
            return union, unionCoverageDepth
        else:
            return union, union.replace(':','^')
    else:
        if version == 'Version2':
            return self.H3K4me3Union, self.H3K4me3UnionCoverageDepth
        else:
            return self.H3K4me3Union, self.H3K4me3Union.replace(':','^')
```

Listing 4.3: getUnion method

accepts the genetic region and statistic version, and uses that to determine which paths are needed for this case. If the statistic is version two, it returns the path to the merged track without overlap, otherwise it will return the path to all the subtypes.

Note that there is an extra test if the region is chromatin. That is because the name of the merged track without overlap for active promoters in chromatin is slightly different than the rest, and all the chromatin states can use the same base paths. It is also worth noticing that in the path to the merged track with overlap, : is replaced with ^, for the same reasons as explained earlier.

When adding another genetic region to the class, it has to be added in this method as well. This is done by adding another test, checking for this new region and returning the paths according to the tracks for this region.

The batchlines

All the different batch lines are put into separate methods, returning a list of strings containing the batch commands. These batch commands vary in size, but the structure is similar. Following is a small example of on how a method may look:

```
1 def batchEnrichBinary(self):
2     return ['@bins =%s'%self.traitTrack, '@INs=', '@INunion=%s'%self.↩
          union,\
3             'hg19|%s|@bins|@INs|@INunion|DiffRelCoverageStat('%self.↩
                  trackType +\
4             'globalSource=chrs,tf1=TrivialFormatConverter,tf2=↩
                  TrivialFormatConverter)']
```

Listing 4.4: Example of batch commands in method

This is the batch line for finding the enrichment with binary normalization. As can be seen, it inserts the different variables at the needed places, and it uses the variable union. That method does not need to do any testing for genetic region, due to the method getUnion earlier. Also notice that the element in position 1 in the list is not set. That empty variable is always positioned at index 1, and will back in the tool be pointing to one and one subtype for the actual genetic region. The position will always be index one to avoid being dependent of the name of the variable.

By adding another method with an appropriate name, that returns the batch command list, it is possible to add more analyses. What needs to be remembered is to make the trait track and merged track path dependent on the variables in the *init* method and not static. And to have the variable that will hold all subtypes at index 1.

The generateBatchLines method

This method is the method that actually returns the correct batch commands. It is called from the tool after all the booleans have been set. Then it goes through a set of if else tests and calls the methods holding the appropriate batch commands. The commands are added to a dictionary, with the keys as the name of the analysis. It returns a dictionary containing lists with strings. Each key has one list, each list is the batch command to one analysis, and each list contains several strings (as seen in listing 4.4).

When adding new analyses to the tool, new tests have to be added to this method. The new tests will have new booleans, and if they are true, it should call the method returning the commands. The commands should be added to the dictionary under an appropriate unique key.

4.2 Genomic track overlap algorithm

Some of the analyses in the tool normalize the result based on local results from all subtypes. As explained in section 4.1.2 and seen in listing 4.1, every test region originally needed two tracks. One containing all segments with overlap, and one containing no overlap but where each segment is valued.

Each genetic region has several subtypes and during hypothesis testing it is needed to look at all of those as one. By merging all subtypes together, allowing overlap and sorting on the start position of each segment, you get what is here called the union track. Although strictly speaking it is not a

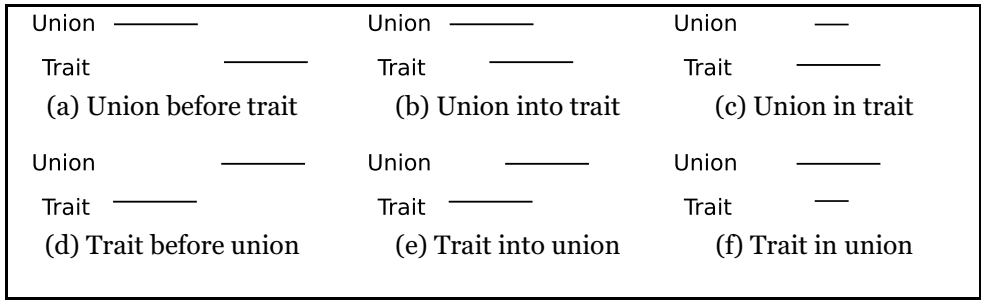


Figure 4.1: The different options for a trait segment to overlap with a union segment. All possible overlaps for the algorithm are these six or variations of this.

union, as it is just appending or merging all the tracks together to one since overlap is allowed.

A flattened version of that track with valued segments, is then called the coverage track. The weighting is found by counting the number of overlapping segments at each position. Figure 3.1d gives a visual explanation on how a weighted track can be viewed.

The single track without overlap is called trait track. The HyperBrowser is written so that it is possible to request all segments from a track, chromosome by chromosome, as a $startsAsArray$ and a $stopsAsArray$. Both these arrays are of the type *numpy.ndarray*. One array holds all the starts positions, and is sorted. The other array holds the stop positions to the same segments, not inclusive. This gives that if the second segment goes from 10 to 15 (inclusive), it will be stored as 10 at index 1 in the array with start positions and 16 at index 1 in the array with the stop position. The union track has the start values sorted, but the end values may not be sorted as it allows for overlapping segments. The trait track will have sorted start values and end values, as it does not contain overlap.

4.2.1 The unwanted track

Originally some of the analyses used the coverage track to do the needed calculations. This is not optimal, as this coverage track has to be pre-produced for each genetic region. This is a computationally heavy and slow process, but it only has to be done once as the track then is stored on disk. Reading a track from disk is reasonably fast. There also exists a method for creating this track if it does not exist, but it runs very slow. The coverage does not have overlapping segments, but instead it contains valued points and segments.

4.2.2 The goal

To optimize the hypothesis testing and make it independent of such a pre-made track, a new algorithm had to be made. The new algorithm should find the overlap between two tracks with segments, one track allowing overlap between the segments (the union track), and one track not allowing

overlap (trait track). This means that the overlap algorithm is to take in four arrays as input, one containing the start positions of the trait track, one containing the end positions for the trait track, one for the start positions of the union track and one containing the end positions for the union track, and return the total overlap between those tracks. This would take away the need for the coverage track, and make it easier to implement new genetic regions as they would not need to create such a track.

4.2.3 Implementing the new algorithm

Setting up the tests

The first task was to set up some test cases using *nosetests*. Nose is a handy tool for writing and managing unit testing in python. It is ran by calling nosetests, where it then scans the current directory for any filenames starting with test. Then it calls all the methods starting with test in those files. It also comes with handy assertion tests. The test file has to import the program or method it is to test and *nose*. For the tests ran here *nose.tools* was chosen, as that includes some more assertion tests, such as tests for checking if equal, almost equal (which lets you set allowed error), and for errors (checks if the correct error is raised). Below is an example of a test written to see if the algorithm handles a union segment starting and ending before the trait segment, as shown in fig. 4.1a. It looks like this:

```
1  def test_beforeTrait():
2      """Union completly outside trait, before"""
3      union = [[1],[4]]
4      trait = [[5],[7]]
5      uStart = array(union[0])
6      uStop = array(union[1])
7      tStart = array(trait[0])
8      tStop = array(trait[1])
9      overlap = countOverlap.overlapStat(uStart, uStop, tStart, tStop)
10     nt.assert_equal(0, overlap)
```

Listing 4.5: Nosetest example

nose.tools is imported as *nt*, and *assert_equal* is used to make sure that the method actually returns zero. The *assert_equal* also takes an optional variable, a string, as a message that will be printed if the assertion fails. The variables union and trait are 2D lists, where index 0 is the list containing starting positions for segments, and index 1 contains the end position for the segments of that index. In this case both contain only one segment. Then the lists are changed to arrays, and sent to the method that will count the overlaps. It then compares the returned result with what is known as the correct number.

There are only a limited number of different ways two segments can overlap; these six are shown in fig. 4.1. And any type of overlap between two tracks containing more segments, will therefore be a combination of these. For example can a segment in the union track overlap the last part of a segment from the trait track and the first part of the next segment. That would be a combination of fig. 4.1e and fig. 4.1b. The first tests written

were therefor tests for the six simple overlaps for two segments, and then expanded for the more obvious combinations of these cases. In the end it 17 tests to make sure all cases were covered.

The test files

For simplicity, the algorithm was not constructed on the HyperBrowser, but on a local desktop. To be able to monitor the efficiency and correctness of the algorithm, three tracks were downloaded from the HyperBrowser to do the tests on. All these files were downloaded as *bed* files, such that the first line contains information about what track it is, and the rest contain the segments. The information is stored column-wise, such that the first column contains the name of the chromosome, the second column contains the starting position of that segment, which is inclusive, the third column the stop position, which is not inclusive, and the fourth column contains a name.

The three main files used were two trait tracks and one union track. The trait tracks were alzheimer SNPs and alzheimer SNPs with 50kb flanks. The segments with 50kb flanks are 100001 long and the SNP are 1 long. The SNPs are just a segment of length one, so for the algorithm it changes nothing. The alzheimer files consist of 41 segments.

The union file is all DHS regions appended together into one, so that several of the segments may overlap each other. It holds 24046672 segments of varying length.

The algorithm

The first version of the overlap method was written as simple and straight forward as possible. The goal was just to get a something working, something that could be improved later. It went through each segment in the trait track, and checked each number between the start position and the end position to see if it was a part of a segment in the union track. This means that for m segments in a trait track of length k, and n segments in the union track, it has to do $m * k * n$ iterations to find the overlap. Considering that k can be anything from one to many thousands, and n can be several millions, this is too inefficient. This demands three loops, where the outer one is for the trait segments, the middle one is for each step in that segment, and the third is for each segment in the union track. To increase efficiency there was added a variable remembering indexes, this variable would remember where to start the search for overlap in the union track. It is know that the trait track is sorted and not overlapping, and the union track is also sorted, and thus it is easy to check if the beginning of a segment from the trait track is larger than the ending on the segment from the union track (as seen in fig. 4.1a). This variable is controlled by a flag, the flag remembers if all previous endings have been smaller than the current start position in the trait segment. If it is, the next index is stored, and the next search starts from that index, ignoring the segments before. This gave the correct answer for all the test cases, but when tried on a real trait track

with all subtypes of DHS merged into the union track and alzheimer SNP with 50kb as trait track, it took too long.

On the test computer (described in section 5.2) it took more than 300 seconds just for chromosome 5, while chromosome 4 took more than 700 seconds. Even if the test was for one chromosome only, this would not be fast enough. Thus one loop had to be removed.

Since the segments are continuous, and based on start and stop position, there is no need to loop through each position in the segment. Since the stop position is not inclusive, the length of a segment is easily calculated as explained in section 3.1. In fig. 4.1 it is shown what options have to be considered when finding out what start position to subtract from what stop position. This is done by, for each segment in the trait track, looping through the segments in the union track and comparing the segments with some tests.

The first thing to check is if the start position in a trait segment overlaps in some way with a segment from the union track. If it does, it is subtracted from the stop position that has the lowest value to find the size of the overlap. This is shown in figs. 4.1b and 4.1f. If the start does not overlap, then the stop position is checked for overlap with the segment from the union track. If overlapping and the start being smaller than the start from the union track segment, then the start from the union track segment is subtracted from the stop position to give size of this overlap. This is the same as fig. 4.1e. The final possible overlap is for union segment completely inside the trait segment. It is checked that the start position for the union segment is larger than the trait segment, and that the stop position for the union segment is smaller than the trait segment. If true, then the size is found. That is fig. 4.1c. The last option left then is for fig. 4.1d, that is checked by seeing if the start of the union segment is larger than the end of the stop position for the trait segment. That would mean that the rest of the union segments will be larger than the current trait segment, and hence the inner loop is broken, and we start over with the next trait segment. Remember that the index for the first union segment to overlap with the trait segment from previous round is stored, which means that the loop does not have to start from the beginning this time. Whenever any overlap is found, it is added to a variable keeping track of the total overlap for that chromosome. To save memory and increase speed, $xrange$ has been used for the generating the list of indexes looped over. This method has the benefit that it does not create the entire list of numbers, but generates a object that returns the numbers in the list [1]. This saves memory and increases speed for large lists such as the ones used here. The method returns an integer which is the overlap counted.

Cython implementation

Cython is a tool for translating Python code into efficient C code. It increases the speed drastically, and it is very efficient for methods doing big iterations. In a way, Cython works really well with the Pareto Principle, or 80/20 rule, that about 80% of the runtime is used in about 20% of the

```
1 python setup.py build_ext --inplace
```

Listing 4.6: Running the script compiling Cython

code[4]. This means that 80% of the code can use Pythons flexible and easy-to-read syntax, and then Cython can be used for the 20% of the code that would benefit from increased speed. For Cython to have the most optimal run time, all the types have to be specified in the code, Cython is typesafe. This is for variables created inside the Cython code, and preferably for input variables. Input variables do not have to be type specific, but that will affect the runtime as it is then done by the Python interpretor [4].

The file extension used for Cython is .pyx and this file is compiled into C code. After compilation it can be imported and used like any other Python library. Compilation is made easy by running a small Python script. The script can be seen in listing A.2, and is run with the command shown in listing 4.6.

The method described in previous section, is a method that would benefit a lot from a Cython implementation. It takes in four arrays, and returns the overlap by looping through the arrays while calculating the overlap of each segment. This method was then pulled out into a .pyx file, as is standard. To make a Cython implementation, the right packages have to be imported, just as with numpy. For this specific implementation numpy had to be imported, and the C version of numpy by doing cimport, the full code can be seen in listing A.1. In Python there is automatic boundary checks for loops over arrays and lists, while in C there is not. Python does also support the use of negative indexes for counting backwards through the list or array, so that index -1 is the last element and index -2 is the seconds last element. Cython has implemented support for both these features, but at the cost of some efficiency. In this implementation these features are not needed and as a consequence they are turned off. For Cython to really speed up the code, it needs explicit variable types. It can handle undeclared variable types as input, but again at a loss of efficiency. This method takes in four one-dimensional arrays with integers, a start and a stop array for each track. In Python one does not need to think about what type of integers the arrays hold, if it is int32 or int64. Cython is typesafe and hence this can become a problem. The creators of the HyperBrowser have not been consistent in the use of data types for the arrays, most are int32 and a few are int64. To maintain maximum speed, this made it necessary to check the datatype of the arrays before sending them into the Cython method. If the array is int64 it is recast to int32. The overlap method will return an integer which is the overlap between all segments checked.

Part III

Results and discussion

Chapter 5

Results

The coming sections about the tool will describe the visuals and include a step-by-step guide on how to use it. This guide will also be available under *Published Pages* on the HyperBrowser. It will also be explained how the tool presents the results and what information the different elements hold. In the final section about the tool, the results from the comparison of MS with B-cells will be presented.

The section about the new algorithm will look at the efficiency of the algorithm and how it compares to the other options already available in the HyperBrowser.

5.1 Genomic variation analysis tool

The tool is generated with a graphical user interface to make it easy to use. The user need not have any programming knowledge as it is all point and click with the mouse. It uses drop-down lists for selections where the user can only select one option, and checkboxes for selections when the user can select more than one option. Submenus, such as normalization type for enrichment are hidden from user until enrichment is selected. This is to avoid filling the interface with choices the user does not need to think about.

It lets the user upload a track with genetic information, and run different comparisons between it and genetic regions. The regions to select between are; DHS, TF, H3K4me3 and the chromatin states active promoter, 4 strong enhancer and 5 strong enhancer. The analyses to choose between are regions touched, enrichment, hypothesis testing, count points and average segment length.

5.1.1 Uploading data

The Genomic HyperBrowser supports uploading tracks through the Galaxy interface it is built upon. The link for that is found under *Galaxy Tools*, and is called *Get data*. For this example the MS confirmed SNPs are stored in bed format in a plain text file on the computer. The bed file is found in the tar.gz file on invitro (see section 1.3). The bed format is needed for this tool.

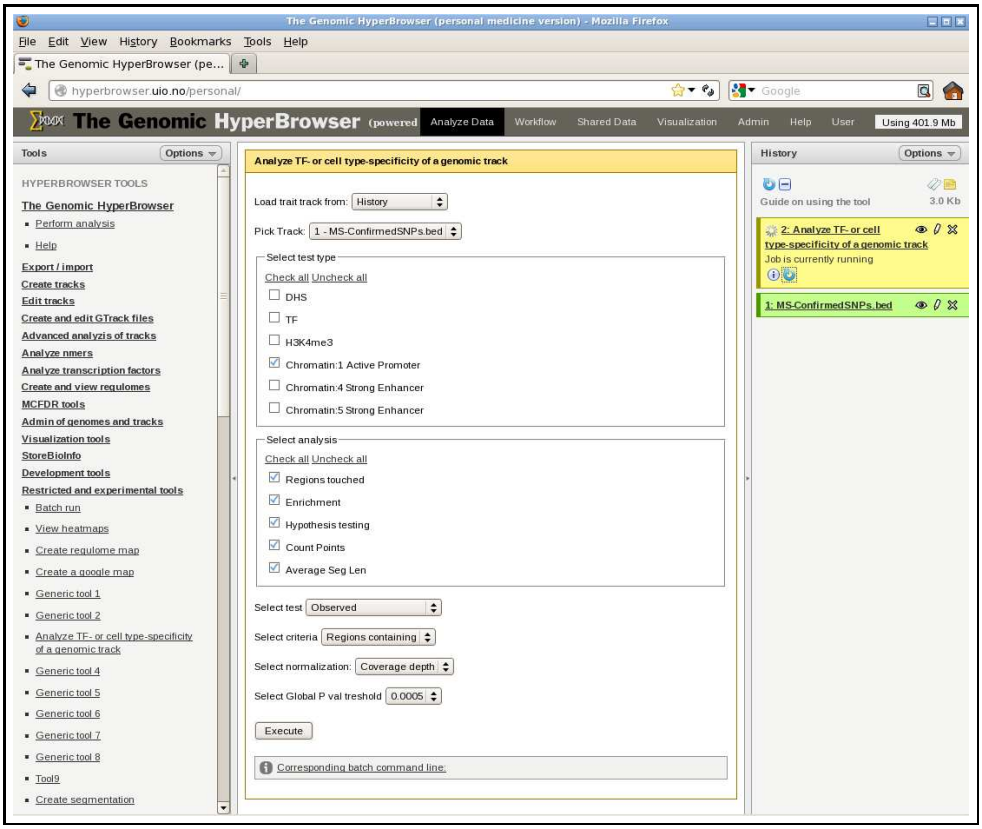


Figure 5.1: Example of how the tool looks, and how things can be selected. Left panel are links to different menus, middle panel is the tool, and right panel contain the history elements. Green history element is finished tasks, yellow is running tasks, and red is failed tasks.

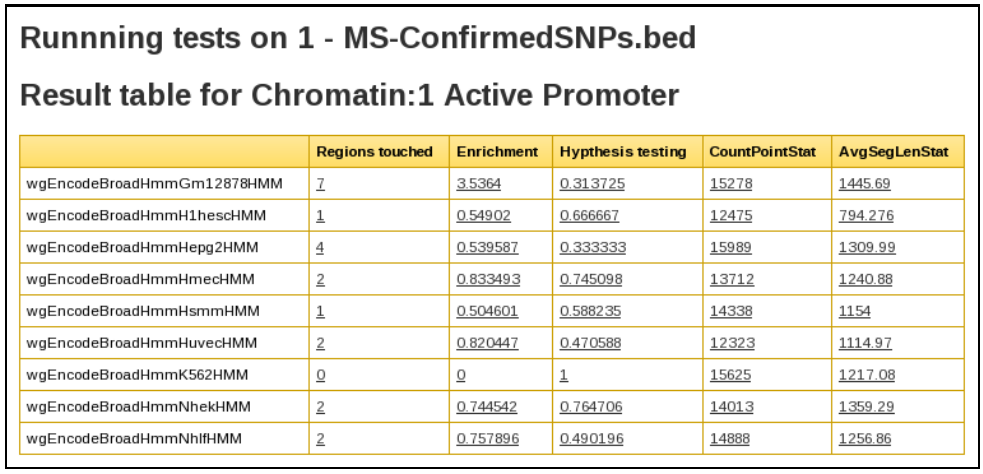
When hitting the *Upload File* link a new page is shown, there one selects the file from local disc and sets file format to bed. The rest is left as it is. When the execute button is hit, the file is uploaded to the Genomic HyperBrowser ready to use.

5.1.2 Using the tool

The file is now stored as an element in the history on the right side panel, and the tool can be selected from the menu on the left side. It is found under *Restricted and experimental tools* and is named *Analyze TF- or cell type-specificity of a genomic track*. This opens the tool in the middle panel. One has to select where to get the track from. In this case the track is uploaded to history, so history is selected. This opens a new drop-down box where the tracks saved to history can be selected. Select the track wanted.

Now it is time to decide which genetic regions to run this track against. For this example the chromatin state active promoter has been selected. But the user can choose more than one if wanted. Each region adds a new set of subtypes, which in turn increases runtime.

The next choice is to select which analyses to run. For this example



Runnning tests on 1 - MS-ConfirmedSNPs.bed

Result table for Chromatin:1 Active Promoter

	Regions touched	Enrichment	Hypthesis testing	CountPointStat	AvgSegLenStat
wgEncodeBroadHmmGm12878HMM	7	3.5364	0.313725	15278	1445.69
wgEncodeBroadHmmH1hescHMM	1	0.54902	0.666667	12475	794.276
wgEncodeBroadHmmHepg2HMM	4	0.539587	0.333333	15989	1309.99
wgEncodeBroadHmmHmecHMM	2	0.833493	0.745098	13712	1240.88
wgEncodeBroadHmmHsmmHMM	1	0.504601	0.588235	14338	1154
wgEncodeBroadHmmHuvecHMM	2	0.820447	0.470588	12323	1114.97
wgEncodeBroadHmmK562HMM	0	0	1	15625	1217.08
wgEncodeBroadHmmNhekHMM	2	0.744542	0.764706	14013	1359.29
wgEncodeBroadHmmNhlfHMM	2	0.757896	0.490196	14888	1256.86

Figure 5.2: Image of a result page for the GWAS Tool

all the analyses are chosen. Normally that would make the tool run for some time, but since there are so few subtypes in active promoter, it is fast enough.

The final step is to select the preferences for the analyses. What box belongs to what is discussed in section 3.2. The preferences chosen are observed, regions containing, coverage depth and 0.0005. Figure 5.1 shows how it should look like after making these selections.

On the right side one can see the history elements. The first one is uploading the MS confirmed SNPs to the HyperBrowser, it is green because it finished without errors. The second history element is yellow, which means it is a task running. It is created when the execute button is clicked, and it will stay yellow for as long as the calculations take. It will turn red if something goes wrong, or green if it finishes without errors.

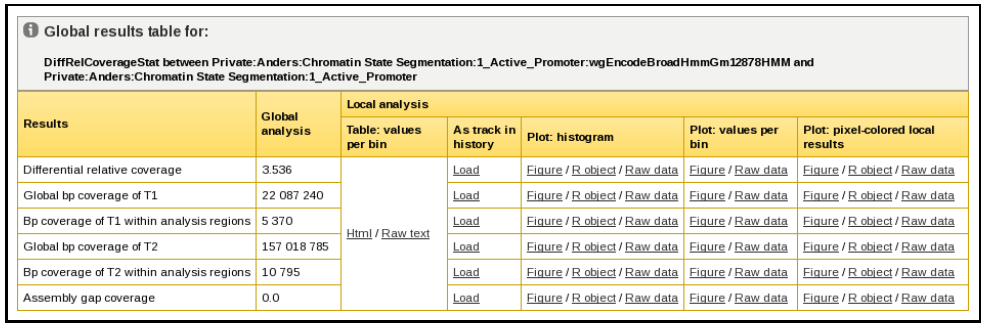
5.1.3 Getting the results

One can click at the eye symbol on the history element to see the results when it turns green. It will show a table containing the results for all the subtypes, the table for this run can be seen in fig. 5.2. Under the table, two graphs will be created. The first shows the enrichment as a bar plot and the P value on top as a scatter plot. Figure 5.5 is the graph from this run. The second graph plots the regions touched, and the result is seen in fig. 5.6.

5.1.4 Presenting the results

The table

When the tool is done with the analyses, it generates a web page with the results. The first things shown are two headers, one saying what track has been analysed, and the second saying for what region the coming results are for. This way the information will always be printed in this order, from top to bottom:

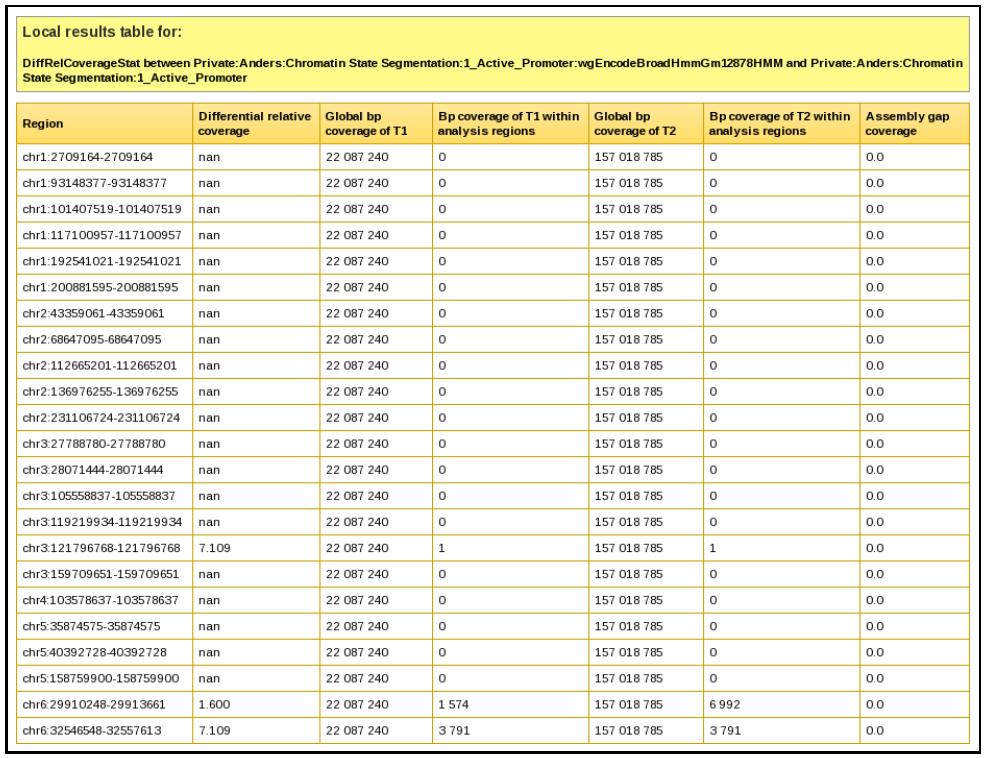


Global results table for:

DiffRelCoverageStat between Private:Anders:Chromatin State Segmentation:1_Active_Promoter:wgEncodeBroadHmmGm12878HMM and Private:Anders:Chromatin State Segmentation:1_Active_Promoter

Results	Global analysis	Local analysis				
		Table: values per bin	As track in history	Plot: histogram	Plot: values per bin	Plot: pixel-colored local results
Differential relative coverage	3.536	Html / Raw text	Load	Figure / R object / Raw data	Figure / Raw data	Figure / R object / Raw data
Global bp coverage of T1	22 087 240		Load	Figure / R object / Raw data	Figure / Raw data	Figure / R object / Raw data
Bp coverage of T1 within analysis regions	5 370		Load	Figure / R object / Raw data	Figure / Raw data	Figure / R object / Raw data
Global bp coverage of T2	157 018 785		Load	Figure / R object / Raw data	Figure / Raw data	Figure / R object / Raw data
Bp coverage of T2 within analysis regions	10 795		Load	Figure / R object / Raw data	Figure / Raw data	Figure / R object / Raw data
Assembly gap coverage	0.0		Load	Figure / R object / Raw data	Figure / Raw data	Figure / R object / Raw data

Figure 5.3: Sub result page for enrichment of MS confirmed in B-cells



Local results table for:

DiffRelCoverageStat between Private:Anders:Chromatin State Segmentation:1_Active_Promoter:wgEncodeBroadHmmGm12878HMM and Private:Anders:Chromatin State Segmentation:1_Active_Promoter

Region	Differential relative coverage	Global bp coverage of T1	Bp coverage of T1 within analysis regions	Global bp coverage of T2	Bp coverage of T2 within analysis regions	Assembly gap coverage
chr1:2709164-2709164	nan	22 087 240	0	157 018 785	0	0.0
chr1:93148377-93148377	nan	22 087 240	0	157 018 785	0	0.0
chr1:101407519-101407519	nan	22 087 240	0	157 018 785	0	0.0
chr1:117100957-117100957	nan	22 087 240	0	157 018 785	0	0.0
chr1:192541021-192541021	nan	22 087 240	0	157 018 785	0	0.0
chr1:200881595-200881595	nan	22 087 240	0	157 018 785	0	0.0
chr2:43359061-43359061	nan	22 087 240	0	157 018 785	0	0.0
chr2:68647095-68647095	nan	22 087 240	0	157 018 785	0	0.0
chr2:112665201-112665201	nan	22 087 240	0	157 018 785	0	0.0
chr2:136976255-136976255	nan	22 087 240	0	157 018 785	0	0.0
chr2:231106724-231106724	nan	22 087 240	0	157 018 785	0	0.0
chr3:27788780-27788780	nan	22 087 240	0	157 018 785	0	0.0
chr3:28071444-28071444	nan	22 087 240	0	157 018 785	0	0.0
chr3:105558837-105558837	nan	22 087 240	0	157 018 785	0	0.0
chr3:119219934-119219934	nan	22 087 240	0	157 018 785	0	0.0
chr3:121796768-121796768	7.109	22 087 240	1	157 018 785	1	0.0
chr3:159709651-159709651	nan	22 087 240	0	157 018 785	0	0.0
chr4:103578637-103578637	nan	22 087 240	0	157 018 785	0	0.0
chr5:35874575-35874575	nan	22 087 240	0	157 018 785	0	0.0
chr5:40392728-40392728	nan	22 087 240	0	157 018 785	0	0.0
chr5:158759900-158759900	nan	22 087 240	0	157 018 785	0	0.0
chr6:29910248-29913661	1.600	22 087 240	1 574	157 018 785	6 992	0.0
chr6:32546548-32557613	7.109	22 087 240	3 791	157 018 785	3 791	0.0

Figure 5.4: Sub result page for each chromosome for the enrichment of MS confirmed in B-cells

1. Header containing track name
2. Header containing genetic region
3. Table holding all global results
4. Graph based on hypothesis testing and enrichment if calculated, fig. 5.5
5. Graph based on regions touched if calculated, fig. 5.6
6. If more regions, start over from item 2

The table holds the global results from one genetic region, where the global results are the results from the entire genome and not just one chromosome. The first column holds the name of all subtypes, and the rest of the columns represent a separate analysis, as can be seen in fig. 5.2. This table has the possibility to sort the results based on each column. By clicking on the header of a column, that column gets sorted highest number and down. Click again and the lowest number comes to the top. The table is created to make it easy for users to find what subtypes to look closer at.

Each number is a link, pointing to a page holding more information about that analysis. What that page presents will be slightly different depending on which analysis it is. Figure 5.3 shows the subpage for MS enrichment in B-cells. If interested one can also get information down to chromosome level by clicking the values per bin link. Figure 5.4 shows a part of the table being shown then.

The graphs

There will be up to two graphs presented to the user, depending on what analyses were selected. Hypothesis testing and enrichment makes for one graph, while regions touched makes for another. Both of these graphs will show the 20 subtypes with the best results, or all subtypes if there are less than 20. The subtypes will be sorted from left to right, low to high.

If hypothesis testing or enrichment has been analysed, there will be a graph under the table presenting the results. That graph will look slightly different depending on which of the analyses has been run. It will show the subtypes with the highest enrichment or lowest p-value. If both have been run, it will show both enrichment and p-value. This makes it easy to see if the enrichment may be significant or not. The graph with both p-value and enrichment can be seen in fig. 5.5. Enrichment is the bars and p-value is the dots. That is also the case even if one of the analyses have been run.

The second graph will show the subtypes with best results for regions touched. Figure 5.6 is an example of how it looks, and this will only be created if regions touched is calculated.

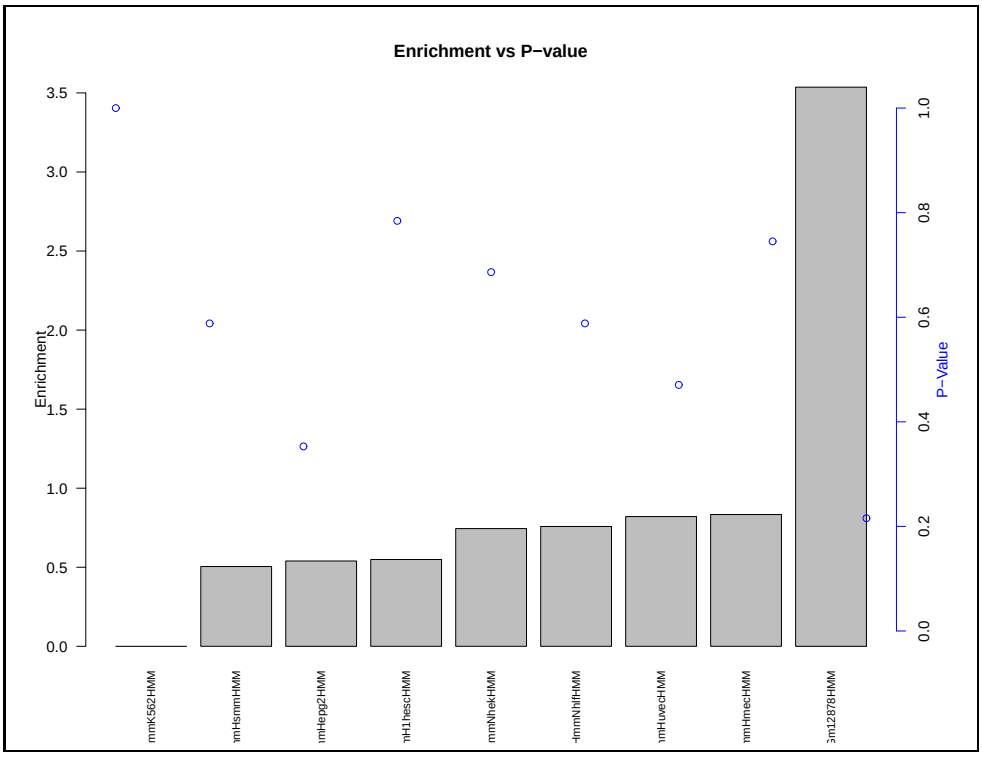


Figure 5.5: Enrichment of MS confirmed SNPs in active promoters

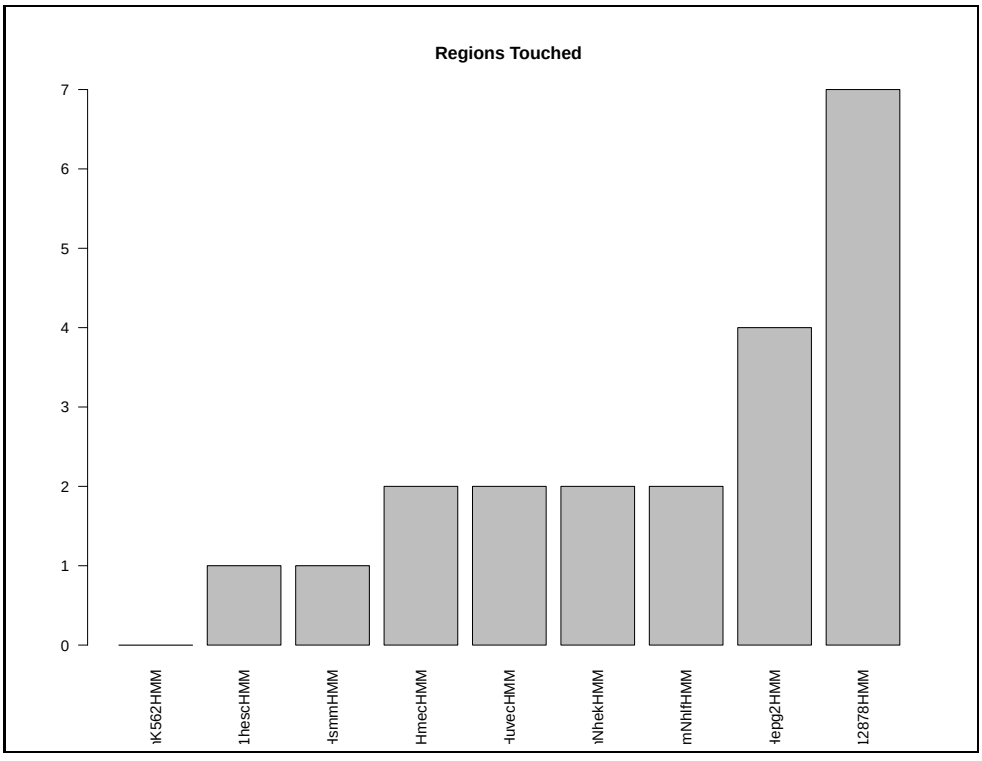


Figure 5.6: Regions touched for MS confirmed SNPs on active promoter

5.1.5 Comparing the results

In "Genomic Regions Associated with Multiple Sclerosis Are Active in B Cells" [13] it was shown that MS SNPs are enriched in B-cells, and that the enrichment is higher than what can be expected by chance. When running MS confirmed in the tool, and comparing it against chromatin states, the results become the same. Figure 5.2 shows that HmmGm12878HMM, B-cells, has the highest regions touched and the highest enrichment with 3.5. The rest of the subtypes have an enrichment between zero and one. B-cells also have the lowest p-value on hypothesis testing. These three results give a strong indication that MS variants active in B-cells, which is the same conclusion as the article came to.

5.2 Genomic track overlap algorithm

When the runtime is kept vague in the following paragraph it is done intentionally. It is not meant to be highly accurate. The time is measured on a computer running Red Hat Linux 4. The processor is an Intel(R) Core(TM) i7 CPU running at 2.93GHz. The time is measured using *time.clock* in Python. Time is kept vague because the needed precautions have not been taken to be sure the times are accurate. There may be big time intervals between the runs, and there are almost always some other programs running in the background that may intervene with how fast the calculations are done. The time is there just to give an indication on how fast or slow it runs.

In section 4.2.3 it was explained how the algorithm was improved by removing one loop and using *xrange*. This increased the speed by huge amounts. Running nosetests on all the tests is done so fast it is irrelevant to time it, less than 0.1 seconds to do all the 17 tests. The algorithm was then tried with the alzheimer with 50kb (alzheimer SNPs with 50000 bases added to each side) file as trait track, and the merged DHS file as union file. The runtime became around 0.9 seconds for chromosome five, and around 0.4 seconds for chromosome four before Cython was implemented, giving the total time across all chromosomes around 11 seconds on the desktop computer.

5.2.1 Timing in HyperBrowser

The overlap method was then implemented into the HyperBrowser framework to see how it compares there. By using the call graph option in HyperBrowser it is possible to get an idea of how efficient the new algorithm is and how it scales for bigger sets. The call graph measures the time in CPU seconds. All of the following tests have been run four times for each algorithm, to increase the chance that the time is accurate. All four times will be presented, but the focus is on the fastest run as that is what is closest to the optimal runtime. Ideally one should run more than four tests to be certain one is close to the optimal time, but the since there is so small differences

Algorithm	Run 1	Run 2	Run 3	Run 4
SumInside	5.951	2.355	**2.269**	2.415
OverlapNoCython	8.236	8.396	8.072	**7.949**
OverlapCython	7.405	8.478	7.423	**7.382**

Table 5.1: Runtime for the different algorithms with MS confirmed with 50kb flanks on DHS regions, measured in CPU seconds

Algorithm	Run 1	Run 2	Run 3	Run 4
OnlyT3	29615.027	29628.287	29600.526	**29520.023**
SumInside	22.653	20.238	19.952	**19.855**
OverlapNoCython	**452.995**	463.951	460.234	475.067
OverlapCython	**134.379**	147.108	146.648	145.897

Table 5.2: Runtime for the different algorithms when doing hypothesis testing with MS confirmed on DHS regions, measured in CPU seconds

between the runs, one can assume it is close enough and that the conclusion would be the same with more runs. The fastest time for each algorithm is boldfaced in the tables. SumInside is the one using the coverage track, OverlapNoCython is the new one without Cython, and OverlapCython is the new with Cython. The MS confirmed used when testing in HyperBrowser contains 61 segments or SNPs. While all DHS subtypes is the same as the union track, with 24046672 segments.

Timing the algorithms

The first tests were to run only the actual algorithms inside the HyperBrowser. In table 5.1 the results from running the algorithms on MS confirmed SNPs with 50kb flanks is presented. The fastest algorithm is SumInside, using only 2.268 seconds. SumInside is also the most limiting algorithm, as it does not allow for overlap, and it needs the coverage track.

The two others algorithms end up almost identical in time. OverlapNoCython has the slowest time as 7.949 seconds, while OverlapCython is slightly faster and finishes in 7.382 seconds. To understand why these two end up being so identical in time, one must go into the call graph and look at time spent in each method. Then one can see that on the fastest run, the actual counting of overlap took 8.42% (0.67 seconds) of the total time. While with OverlapCython it is so fast that the call graph does not present it. But since OverlapCython is type dependent, the method calling the overlap algorithms has to check and maybe recast the types. That increases the time spent in that method from 0.11% to 0.15%. This also shows that the algorithms are reasonably fast, and that the bottleneck probably is something else.

Algorithm	Run 1	Run 2	Run 3	Run 4
OnlyT3	**37746.745**	37826.370	37811.998	37910.405
SumInside	2703.704	2719.715	2708.498	**2697.290**
OverlapNoCython	1861.890	1920.645	**1823.519**	1843.009
OverlapCython	**172.671**	173.990	183.637	174.551

Table 5.3: Runtime for the different algorithms when doing hypothesis testing with CD19 on DHS regions, measured in CPU seconds

Timing it on hypothesis testing

Next it is interesting to see how the algorithms perform in an actual analysis. Hypothesis testing with MS confirmed SNPs on 325 - GM12878 (a subtype of DHS), normalized based on coverage depth. For this, there are four algorithms that can be used to find the overlap, SumInside, OverlapNoCython, OverlapCython and OnlyT3. OnlyT3 is the old algorithm that allowed overlapping segments, and it is used if no coverage depth track exists. There were 50 randomized tries done for this test. The results can be seen in table 5.2. It is clear that the slowest algorithm is OnlyT3 as it used 29520.023 seconds (more than 8 hours). SumInside was by far the fastest one, using only 19.855 seconds. OverlapNoCython used 452.995 seconds (about 7 minutes) and OverlapCython used 134.379 seconds (about 2 minutes). Again it might be interesting to take a look at how much time is used on actually calculating the overlap. For OverlapNoCython it uses 70.11% (about 5 minutes) of the time calculating the overlap, while OverlapCython uses 3.91% (about 5 seconds). A part of the call graph for OverlapCython can be seen in fig. A.1a

Timing it on hypothesis testing, bigger set

To see how this scaled when the trait track becomes big, the same tests were run as in section 5.2.1. But this time it was tried with CD19 as trait track, and checked against 1 - A549. Both of these are subtypes of DHS, which probably makes it biologically uninteresting to do a test like this. But the size of CD-19 makes it algorithmically interesting. CD-19 contains 75086 segments, and is thus much larger than the MS track. The results are presented in table 5.3. OnlyT3 is again the slowest one, using 37746.745 seconds (more than 10 hours). SumInside is now the second slowest using 2697.290 seconds (almost 45 minutes), the OverlapNoCython uses 1823.519 secods (almost 31 minutes). And OverlapCython used 172.671 seconds (almost 3 minutes). By taking a closer look at the two algorithms again, one can see that 91.16% (almost 28 minutes) of the time is used to count the overlap in OverlapNoCython. While with OverlapCython it uses only 7.38% (almost 13 seconds) to count the overlap. A part of the call graph for OverlapCython can be seen in fig. A.1b.

Chapter 6

Discussion

This chapter will consist of three parts. The first part covers how to interact with the HyperBrowser and how to save time when working with it. The next section will cover weaknesses in the tool and the overlap algorithm; why they exist, and what can be done with them. Future work will be discussed in the final section; what more can be done, and what should be done before it is released as a finished tool.

6.1 Legacy Code

In this section some guidelines for adding new functionality to the HyperBrowser will be explained, the examples should work for adding functionality to legacy code in general as well. The guidelines are created based on mistakes made and experience learned while working with the HyperBrowser. Many mistakes and much time could have been saved if these guidelines had been followed when creating the tool. The new functionality can be one or more methods added to the HyperBrowser, without the need to really change existing code.

6.1.1 Collect the right information

The first task after deciding what function to add, should be to collect information. Consider the following questions: Where in the HyperBrowser should it be added? What methods does the new function need to call? What do the methods return? It is also useful to look at what other functions use these methods, this way it is possible to collect information about what comes in and what goes out of those methods. Wrap the methods in print statements, print everything going in, everything going out and also put print statements inside the methods if possible. Then run the functions using these methods, but run them on small cases. The faster a test can finish, the better. If it takes a day to write a smaller test case that takes a few minutes run, when the original case takes a few hours, then it does not take many runs to make it worth it. Eventually the code will start to make more sense, and it will be clearer what all the variables are. Then the time has come to look for testing points.

6.1.2 Finding the points and writing the tests

The first and most obvious place to write tests, are for the variables being returned by the methods already in the HyperBrowser. This makes it clear what variables the new function has to work with from the HyperBrowser code. More generally any part of the new function that interacts with HyperBrowser code should be tested. Any variables going from the new function to the HyperBrowser should be checked, and the variables being returned to the new function should be checked. The tests for variables being sent from the HyperBrowser to the new functionality can be run before the actual functionality is in place, as they should be able to pass. When those are done, it is time to write the tests for the new functionality, it should by now be clear what it should pass to other methods and what it should get back. These tests will, of course, be failing until the methods in the new functionality gets written.

When all the tests are running, then the new functionality should be running properly for at least the cases tested. It may then be useful to make sure that all possible cases are actually covered by a test case.

6.1.3 Make sure examples are correct

If the new functionality is to make use of information, paths, program examples or similar, then make sure they are correct. If the new functionality is supposed be a graphical interface for some analyses run with batchlines, then make sure the batchlines are working before implementing them. It is easier to debug if the batchlines are run through the *Debug batchlines* method in the HyperBrowser, compared to if it is run through another tool. Wait to implement them until they actually run as they should.

6.2 Weaknesses

The following subsections will look at different weaknesses in both the tool and the new overlap algorithm. The weaknesses will be explained, and possible solutions will be presented.

6.2.1 Genomic variation analysis tool

Lack of explanations

One of the first weaknesses one might find is that the tool contains very few explanations. Each selection only contains a name or a short statement. This can make the tool difficult to understand for a user with little or no knowledge about the HyperBrowser, the genetic regions or the analyses. If the user wants to load a track from history, but has no bed tracks in history, then the drop-down list comes up empty without any explanations.

The tool assumes that the user knows what the different genetic regions are; it does not, for example, explain what DHS is or when that is a good

choice. As to the analyses, these to have no explanations. It is assumed that the user knows the difference between enrichment and hypothesis testing, or what regions the average segment length is calculated for.

The reason why no explanation or extra information has been added, is simply because it has not been a priority. The tool is a proof of concept, and thus it will not be released out to many users with no knowledge of this.

Lack of error handling

The tool does no testing to make sure that the users use it correctly. The tool will try to run if the user hits execute without selecting a track, and the history element will turn red due to a crash. Nothing will show up if the link to the results is selected, and the message showing where Python crashed will show if the history element is expanded and the bug is clicked. If a trait track is selected but no genetic regions and no analysis, it will run and turn green. However, there will be no results showing, and no message to the user about why there are no results. If an analysis is selected without any genetic region, the same will happen. Again, if a genetic region is selected, but no analysis, the same will happen as if no track was selected. There is also no checking to make sure that the trait track actually is of the right type. If the user tries to use a trait track with genetic information that is not made for the hg19/GRCh37, the results may be wrong or it might even fail if the information is outside the correct genetic regions.

Once again, this is not implemented, due to this being a proof of concept, which has made that less of a prioritized task.

The code is not flexible enough

Although it is reasonable easy to add new genetic regions to the objects, it is not easy enough to modify the code in general. The code could have been made more flexible, making it easier to expand with new analyses and genetic regions, especially the code for the tool.

Since the user does not see the code of this tool, this has not been the focus. As a proof of concept, it is not important to have perfect code. The user does not see this anyway.

Could have been parallelized

This tool is currently doing everything linear, it does many time consuming independent steps after each other. Since these steps are more or less completely independent, this could be parallelized. Each genetic region could be done in a separate process, then each process would get its own instance of a region object and the path to the trait track. This would probably greatly increase the efficiency, as some of these genetic regions can take many hours to finish if enough analyses is selected.

This has not been implemented for two reasons, firstly the HyperBrowser has a feature for automatic parallelization (currently turned). This means that the work of making the code parallel might become redundant if

the feature is activated again. Secondly, this tool is made to show scientists what can be done; how easy it could be made to do these types of analyses, and parallelization does not improve that.

6.2.2 Genomic track overlap algorithm

Efficiency

The two new algorithms performed equally well when running on MS confirmed with 50kb flanks against DHS. SumInside on the other hand performed much better. This result is interesting, since the OverlapNoCython only used slightly more than half a second in the actual overlap algorithm. And OverlapCython used so little time that it did not even show on the call graph.

The differences become much clearer when using the algorithms on an actual hypothesis test. SumInside is still the fastest for small trait tracks, but when the trait track gets big that changes. MS confirmed contains 61 segments, while CD-19 contains 75086 segments. This means that the number of segments in the trait track was increased almost 1231 fold. For the SumInside the time then increased by 135 fold, while with the OverlapCython it increased only 1.3 fold. These numbers, combined with the call graph showing that 73.1% of the time was spent collecting arrays in the OverlapCython, might mean that to increase efficiency one might benefit more from looking at how the data is collected or the HyperBrowser in general, than at the actual algorithm.

Strict data type

One might think that one weakness of the algorithm is that it is typesafe, and it will be, if the integers start to get bigger than 2147483647 (maximum number for int32) or if they start using int64 as standard. It is a simple task to modify the Cython code to accept int64, or any other type. The small test done to check the type of the arrays, and recasting them if necessary, is also easy to change. The test is written so that only one variable has to be modified to fit the new data type. Since the numbers represent points along the natural line, there is no reason they should change to floats. It would only be a small change in the Cython code and a recompiling of the code, if they chould decide to change the array type. This means that the extra work that might come from Cython being typesafe, is easily beaten by its efficiency.

6.3 Future work

6.3.1 Genomic variation analysis tool

Before too much is changed with the tool, there should be added some more explanations to the tool. So as to make it clear what each selection does. Then it should be introduced to a selected group of scientists. That way

more feedback could be collected, and it would be easier to know how to further design the tool to their needs. One scientist has already tried the tool, and his wishes for further work are more information about which regions are affecting the enrichment value, more information about what the actual analyses does and more regions added to the chromatin states.

There are several things that can be done as further work for this tool. The structure between the tool in the HyperBrowser and the GwasBatchLines class should probably be changed. It is fairly simple to add another genetic region, but it is somewhat more difficult to add a new analysis, but both of these should be improved. This would make the tool more flexible, and easier to modify as the need of the scientists might change with new discoveries.

Depending on what is decided in regards to the parallelization feature, it might be relevant to increase efficiency by parallelizing the code. There is no doubt that it would make the code faster, but it would have to take into account for limited system resources.

6.3.2 Genomic track overlap algorithm

There are mainly two things that can be done as further work. The first is to expand the algorithm, or make a new one, allowing for internal overlap in both tracks. This should probably be done as a new algorithm, as it will affect the efficiency of the algorithm.

The second thing that could be done, is to look at how the HyperBrowser stores and collects the data. Since collecting the data is what takes the most time with the current algorithm.

Chapter 7

Conclusion

The main goal was to develop a proof of concept for an easy to use tool for analysing GWAS data, typically genetic variations. Using this tool the user can upload a track containing information about these variations, and then analyse it against certain genomic regions, looking for cell type specificity. The results were to be presented so that it is easy to find the results of interest.

This tool was created in the Genomic HyperBrowser, with a graphical interface for ease of use. The tool has been tried by one scientist, and the feedback was positive. He is satisfied with how easy the tool is to use. The tool is publicly available at http://hyperbrowser.uio.no/personal/.

A subgoal became to improve efficiency of the more flexible algorithm, so that the tool easier could be expanded to do analyses on different regions. A more efficient algorithm was developed that allows for internal overlap in one of the tracks. It also becomes much faster than the stricter algorithm when the track without overlap becomes large.

In conclusion, the tool became flexible and easy to use, and the algorithm greatly decreased the time it takes to find the overlap between two tracks where one has internal overlapping segments.

Bibliography

Journal papers

[4] S. Behnel, R. Bradshaw, C. Citro, L. Dalcin, D. Seljebotn, and K. Smith. "Cython: The Best of Both Worlds." In: *Computing in Science Engineering* 13.2 (2011). ISSN: 1521-9615. DOI: 10.1109/MCSE.2010.118 (cit. on pp. 14, 48).

[5] D. Blankenberg, N. Coraor, G. Von Kuster, J. Taylor, and A. Nekrutenko. "Integrating diverse databases into an unified analysis framework: a Galaxy approach." In: *Database: The Journal of Biological Databases and Curation* 2011 (2011). ISSN: 1758-0463. DOI: 10.1093/database/bar011 (cit. on pp. 3, 16).

[7] W. S. Bush and J. H. Moore. "Chapter 11: genome-wide association studies." In: *PLoS computational biology* 8.12 (2012). ISSN: 1553-7358. DOI: 10.1371/journal.pcbi.1002822 (cit. on pp. 9, 10, 12).

[8] M. C. Castaño Betancourt, F. Cailotto, H. J. Kerkhof, F. M. F. Cornelis, S. A. Doherty, D. J. Hart, A. Hofman, F. P. Luyten, R. A. Maciewicz, M. Mangino, S. Metrustry, K. Muir, M. J. Peters, F. Rivadeneira, M. Wheeler, W. Zhang, N. Arden, T. D. Spector, A. G. Uitterlinden, M. Doherty, R. J. U. Lories, A. M. Valdes, and J. B. J. van Meurs. "Genome-wide association and functional studies identify the DOT1L gene to be involved in cartilage thickness and hip osteoarthritis." eng. In: *Proceedings of the National Academy of Sciences of the United States of America* 109.21 (2012). ISSN: 1091-6490. DOI: 10.1073/pnas.1119899109 (cit. on p. 11).

[10] J. Cohen, A. Wilson, and K. Manzolillo. "Clinical and economic challenges facing pharmacogenomics." In: *Pharmacogenomics J* (2012). ISSN: 1473-1150 (cit. on p. 13).

[13] G. Disanto, G. K. Sandve, A. J. Berlanga-Taylor, J. M. Morahan, R. Dobson, G. Giovannoni, and S. V. Ramagopalan. "Genomic regions associated with multiple sclerosis are active in B cells." In: *PloS one* 7.3 (2012). ISSN: 1932-6203. DOI: 10.1371/journal.pone.0032281 (cit. on pp. 3, 4, 20, 21, 57).

[14] X. Dong, L. Wang, K. Taniguchi, X. Wang, J. M. Cunningham, S. K. McDonnell, C. Qian, A. F. Marks, S. L. Slager, B. J. Peterson, D. I. Smith, J. C. Cheville, M. L. Blute, S. J. Jacobsen, D. J. Schaid, D. J. Tindall, S. N. Thibodeau, and W. Liu. "Mutations in

CHEK2 associated with prostate cancer risk." eng. In: *American journal of human genetics* 72.2 (2003). ISSN: 0002-9297. DOI: 10.1086/346094 (cit. on p. 3).

[15] I. Dunham et al. "An integrated encyclopedia of DNA elements in the human genome." In: *Nature* 489.7414 (2012). ISSN: 1476-4687. DOI: 10.1038/nature11247 (cit. on p. 20).

[18] A. Gerasimova, L. Chavez, B. Li, G. Seumois, J. Greenbaum, A. Rao, P. Vijayanand, and B. Peters. "Predicting Cell Types and Genetic Variations Contributing to Disease by Combining GWAS and Epigenetic Data." In: *PloS one* 8.1 (2013). ISSN: 1932-6203. DOI: 10.1371/journal.pone.0054359 (cit. on pp. 20, 25).

[19] S. Gundersen, M. Kalaš, O. Abul, A. Frigessi, E. Hovig, and G. K. Sandve. "Identifying elemental genomic track types and representing them uniformly." en. In: *BMC Bioinformatics* 12.1 (2011). ISSN: 1471-2105. DOI: 10.1186/1471-2105-12-494 (cit. on p. 32).

[20] A. E. Handel, G. Disanto, and S. V. Ramagopalan. "Next-generation sequencing in understanding complex neurological disease." In: *Expert review of neurotherapeutics* 13.2 (2013). ISSN: 1744-8360. DOI: 10.1586/ern.12.165 (cit. on pp. 20, 28).

[21] J. Hsu, P. C. Avila, R. C. Kern, M. G. Hayes, R. P. Schleimer, and J. M. Pinto. "Genetics of chronic rhinosinusitis: State of the field and directions forward." eng. In: *The Journal of allergy and clinical immunology* 131.4 (2013). ISSN: 1097-6825. DOI: 10.1016/j.jaci.2013.01.028 (cit. on p. 21).

[22] S. Ikegawa. "A short history of the genome-wide association study: where we were and where we are going." In: *Genomics & informatics* 10.4 (2012). ISSN: 1598-866X. DOI: 10.5808/GI.2012.10.4.220 (cit. on pp. 3, 11, 12).

[23] K. J. Karczewski, R. P. Tirrell, P. Cordero, N. P. Tatonetti, J. T. Dudley, K. Salari, M. Snyder, R. B. Altman, and S. K. Kim. "Interpretome: a freely available, modular, and secure personal genome interpretation engine." In: *Pacific Symposium on Biocomputing*. *Pacific Symposium on Biocomputing* (2012). ISSN: 1793-5091 (cit. on p. 13).

[26] L. J. Lesko. "Personalized medicine: elusive dream or imminent reality?" eng. In: *Clinical pharmacology and therapeutics* 81.6 (2007). ISSN: 0009-9236. DOI: 10.1038/sj.clpt.6100204 (cit. on pp. 12, 13).

[27] F. Liu, B. Wen, and M. Kayser. "Colorful DNA polymorphisms in humans." ENG. In: *Seminars in cell & developmental biology* (2013). ISSN: 1096-3634. DOI: 10.1016/j.semcdb.2013.03.013 (cit. on p. 3).

[28] T. A. Manolio. "Genomewide association studies and assessment of the risk of disease." In: *The New England journal of medicine* 363.2 (2010). ISSN: 1533-4406. DOI: 10.1056/NEJMra0905980 (cit. on pp. 10, 12).

[29] M. T. Maurano, R. Humbert, E. Rynes, R. E. Thurman, E. Haugen, H. Wang, A. P. Reynolds, R. Sandstrom, H. Qu, J. Brody, A. Shafer, F. Neri, K. Lee, T. Kutyavin, S. Stehling-Sun, A. K. Johnson, T. K. Canfield, E. Giste, M. Diegel, D. Bates, R. S. Hansen, S. Neph, P. J. Sabo, S. Heimfeld, A. Raubitschek, S. Ziegler, C. Cotsapas, N. Sotoodehnia, I. Glass, S. R. Sunyaev, R. Kaul, and J. A. Stamatoyannopoulos. "Systematic localization of common disease-associated variation in regulatory DNA." In: *Science (New York, N.Y.)* 337.6099 (2012). ISSN: 1095-9203. DOI: 10.1126/science.1222794 (cit. on p. 20).

[30] S. D. McCulloch and T. A. Kunkel. "The fidelity of DNA synthesis by eukaryotic replicative and translesion synthesis polymerases." eng. In: *Cell research* 18.1 (2008). ISSN: 1748-7838. DOI: 10.1038/cr.2008.4 (cit. on p. 7).

[31] B. Moorefield. "PICking H3K4me3." eng. In: *Nature structural & molecular biology* 20.4 (2013). ISSN: 1545-9985. DOI: 10.1038/nsmb.2561 (cit. on p. 8).

[32] P. C. Ng, S. S. Murray, S. Levy, and J. C. Venter. "An agenda for personalized medicine." In: *Nature* 461.7265 (2009). ISSN: 1476-4687. DOI: 10.1038/461724a (cit. on pp. 13, 14).

[33] T. A. Pearson and T. A. Manolio. "How to interpret a genome-wide association study." In: *JAMA: the journal of the American Medical Association* 299.11 (2008). ISSN: 1538-3598. DOI: 10.1001/jama.299.11.1335 (cit. on p. 10).

[36] G. K. Sandve, S. Gundersen, H. Rydbeck, I. K. Glad, L. Holden, M. Holden, K. Liestøl, T. Clancy, E. Ferkingstad, M. Johansen, V. Nygaard, E. Tøstesen, A. Frigessi, and E. Hovig. "The Genomic HyperBrowser: inferential genomics at the sequence level." In: *Genome Biology* 11.12 (2010). ISSN: 1465-6906. DOI: 10.1186/gb-2010-11-12-r121 (cit. on pp. 15–17).

[37] S. Sanna, A. U. Jackson, R. Nagaraja, C. J. Willer, W.-M. Chen, L. L. Bonnycastle, H. Shen, N. Timpson, G. Lettre, G. Usala, P. S. Chines, H. M. Stringham, L. J. Scott, M. Dei, S. Lai, G. Albai, L. Crisponi, S. Naitza, K. F. Doheny, E. W. Pugh, Y. Ben-Shlomo, S. Ebrahim, D. A. Lawlor, R. N. Bergman, R. M. Watanabe, M. Uda, J. Tuomilehto, J. Coresh, J. N. Hirschhorn, A. R. Shuldiner, D. Schlessinger, F. S. Collins, G. Davey Smith, E. Boerwinkle, A. Cao, M. Boehnke, G. R. Abecasis, and K. L. Mohlke. "Common variants in the GDF5-UQCC region are associated with variation in human height." eng. In: *Nature genetics* 40.2 (2008). ISSN: 1546-1718. DOI: 10.1038/ng.74 (cit. on p. 11).

[38] T. F. Tedder, M. Streuli, S. F. Schlossman, and H. Saito. "Isolation and structure of a cDNA encoding the B1 (CD20) cell-surface antigen of human B lymphocytes." In: *Proceedings of the National Academy of Sciences of the United States of America* 85.1 (1988). ISSN: 0027-8424 (cit. on p. 21).

[39] G. Trynka, C. Sandor, B. Han, H. Xu, B. E. Stranger, X. S. Liu, and S. Raychaudhuri. “Chromatin marks identify critical cell types for fine mapping complex trait variants.” In: *Nature genetics* 45.2 (2013). ISSN: 1546-1718. DOI: 10.1038/ng.2504 (cit. on p. 20).

[40] B. N. Vardarajan, S. Y. Bruesegem, M. E. Harbour, P. St George-Hyslop, M. N. J. Seaman, and L. A. Farrer. “Identification of Alzheimer disease-associated variants in genes that regulate retromer function.” eng. In: *Neurobiology of aging* 33.9 (2012). ISSN: 1558-1497. DOI: 10.1016/j.neurobiolaging.2012.04.020 (cit. on p. 3).

[41] L. D. Ward and M. Kellis. “HaploReg: a resource for exploring chromatin states, conservation, and regulatory motif alterations within sets of genetically linked variants.” eng. In: *Nucleic acids research* 40.Database issue (2012). ISSN: 1362-4962. DOI: 10.1093/nar/gkr917 (cit. on p. 21).

[43] W. Yu, A. Yesupriya, A. Wulf, L. A. Hindorff, N. Dowling, M. J. Khoury, and M. Gwinn. “GWAS Integrator: a bioinformatics tool to explore human genetic associations reported in published genome-wide association studies.” eng. In: *European journal of human genetics: EJHG* 19.10 (2011). ISSN: 1476-5438. DOI: 10.1038/ejhg.2011.91 (cit. on p. 21).

Other written references

[6] R. Brooker, E. Widmaier, L. Graham, and P. Stiling. *Biology*. 2nd ed. McGraw-Hill, 2008. ISBN: 0072956207 (cit. on pp. 7, 8).

[16] M. Feathers. *Working Effectively with Legacy Code*. en. Prentice Hall Professional, Sept. 2004. ISBN: 9780132931755 (cit. on pp. 17–19).

[25] H. P. Langtangen. *Python Scripting for Computational Science*. en. Springer-Verlag GmbH, Mar. 2012. ISBN: 9783642219627 (cit. on pp. 14, 15).

[42] J. Xiong. *Essential Bioinformatics*. en. Cambridge University Press, Mar. 2006. ISBN: 9781139450621 (cit. on pp. 8, 9).

Online references

[1] *2. Built-in Functions - Python v2.7.4 documentation*. URL: http://docs.python.org/2/library/functions.html#xrange (visited on 04/12/2013) (cit. on p. 47).

[2] *8.3. collections - High-performance container datatypes - Python v2.7.4 documentation*. URL: http://docs.python.org/2/library/collections.html#collections.OrderedDict (visited on 04/28/2013) (cit. on p. 38).

[3] *B cell definition - Medical Dictionary definitions of popular medical terms easily defined on MedTerms*. Apr. 27, 2011. URL: http : // www.medterms.com/script/main/art.asp?articlekey=2413 (visited on 04/24/2013) (cit. on p. 21).

[9] *Cholesterol, LDL definition - Cholesterol Information Produced by Doctors For Patients Experiencing High Cholesterol Levels*. Mar. 30, 2012. URL: http://www.medterms.com/script/main/art.asp?articlekey=2714 (visited on 04/27/2013) (cit. on p. 21).

[11] *Complementary DNA definition - Medical Dictionary definitions of popular medical terms easily defined on MedTerms*. June 14, 2012. URL: http://www.medterms.com/script/main/art.asp?articlekey=2806 (visited on 04/27/2013) (cit. on p. 21).

[12] *de novo - definition of de novo by the Free Online Dictionary, Thesaurus and Encyclopedia*. URL: http : //www.thefreedictionary.com/de+novo (visited on 04/28/2013) (cit. on p. 21).

[17] *Fibroblast definition - Medical Dictionary definitions of popular medical terms easily defined on MedTerms*. Mar. 19, 2012. URL: http : //www.medterms.com/script/main/art.asp?articlekey = 24766 (visited on 04/24/2013) (cit. on p. 21).

[24] *Keratinocyte - Medical Definition and More from Merriam-Webster*. URL: http://www.merriam-webster.com/medical/keratinocyte (visited on 04/27/2013) (cit. on p. 21).

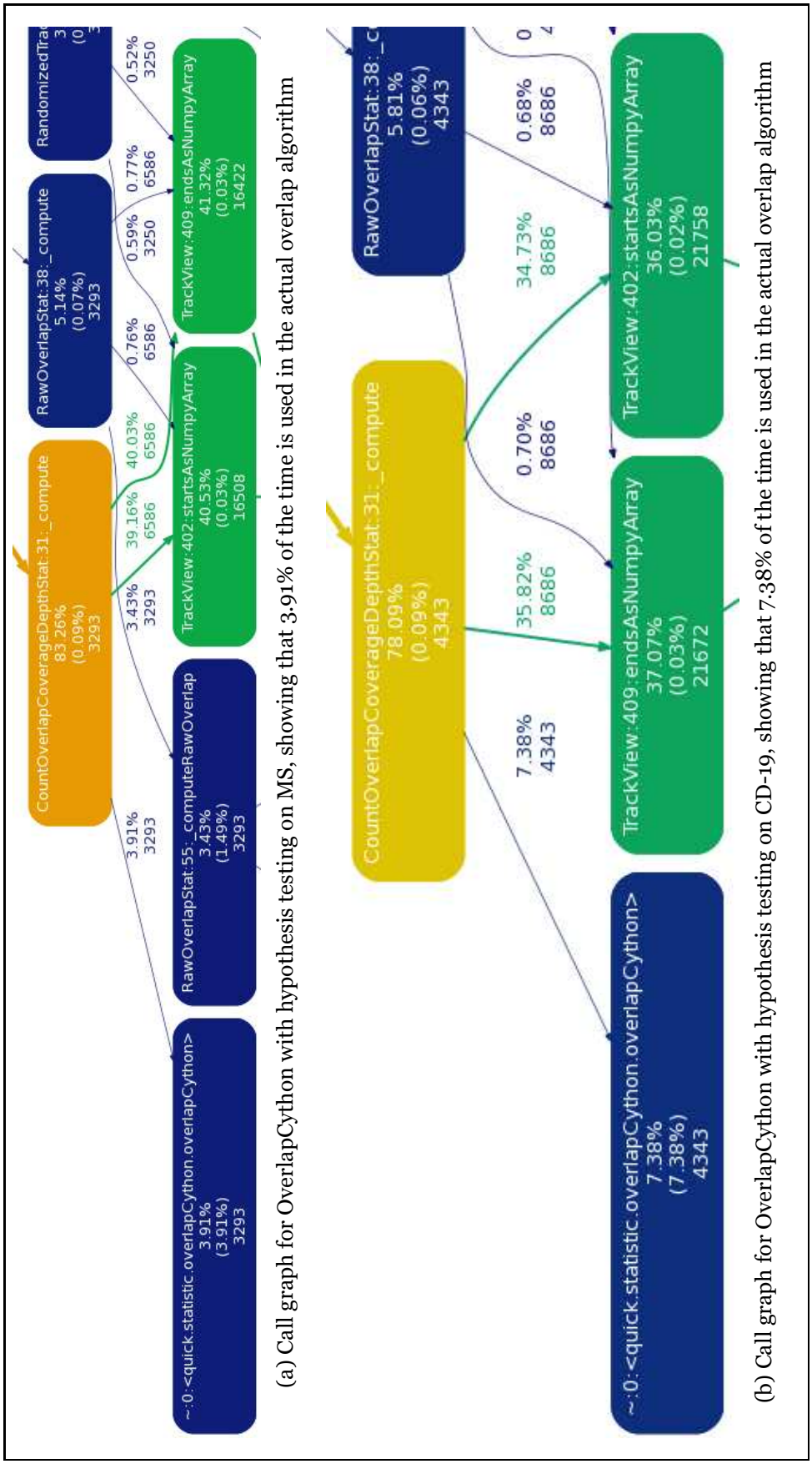
[34] *Primary Biliary Cirrhosis - Complete medical information regarding this chronic disease on MedicineNet.com*. URL: http : // www.medicinenet.com/primary_biliary_cirrhosis/article.htm (visited on 04/24/2013) (cit. on p. 21).

[35] *Rheumatoid Arthritis Treatment, Diet, Medications, Diagnosis - MedicineNet*. Apr. 16, 2013. URL: http : //www.medicinenet.com/rheumatoid_arthritis/article.htm (visited on 04/27/2013) (cit. on p. 21).

Appendix

Appendix A

Cython



(a) Call graph for OverlapCython with hypothesis testing on MS, showing that 3.91% of the time is used in the actual overlap algorithm

(b) Call graph for OverlapCython with hypothesis testing on CD-19, showing that 7.38% of the time is used in the actual overlap algorithm

```
import numpy as np
cimport numpy as np
cimport cython
from cpython cimport bool
ctypedef np.int64 DT

@cython.boundscheck(False)  # turn off array bounds check
@cython.wraparound(False)   # turn off negative indices (u[-1,-1])
cpdef overlapCython(np.ndarray[DT, ndim=1] traitStart,
        np.ndarray[DT, ndim=1] traitStop,
        np.ndarray[DT, ndim=1] unionStart,
        np.ndarray[DT, ndim=1] unionStop):
        cdef int traitLen, unionLen
        cdef int tmpInd, i, k,
        cdef int count, start, stop
        cdef bool flag
        traitLen = traitStart.size
        unionLen = unionStart.size

        count = 0
        tmpInd = 0
        for i in xrange(traitLen):
            start = traitStart[i]
            stop = traitStop[i]
            indStart = tmpInd
            flag = True
            for k in xrange(indStart, unionLen):
                if start >= unionStop[k] and flag:
                    tmpInd = k+1
                elif start >= unionStart[k] and start < unionStop[k]:
                    #if traitstart inside union sequence
                    flag = False
                    if stop <= unionStop[k]:
                        count += stop-start
                    else:
                        count += unionStop[k]-start
                elif stop > unionStart[k] and stop <= unionStop[k]:
                    flag = False
                    if start <= unionStart[k]:
                        count += stop - unionStart[k]
                elif unionStart[k] >= start and unionStop[k] <= stop:
                    flag = False
                    count += unionStop[k] - unionStart[k]
                elif stop <= unionStart[k]:
                    flag = False
                    break
        return count
```

Listing A.1: Cython code of the overlap method before compilation

```
1  from distutils.core import setup
2  from distutils.extension import Extension
3  import numpy as np
4  from Cython.Distutils import build_ext
5
6  cymodule = 'overlapCython' # Name of cython file
7  setup(
8          name=cymodule
9          ext_modules=[
10             Extension(cymodule, [cymodule + '.pyx'],
11             include_dirs=[np.get_include()])],
12         cmdclass={'build_ext': build_ext},
13         )
```

Listing A.2: Setup file compiling Cython code